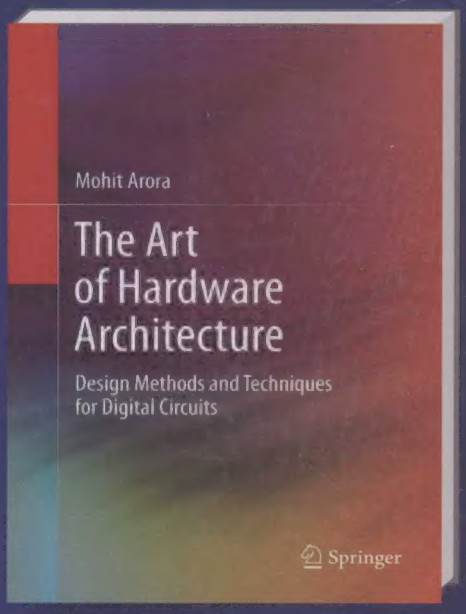
HZ BOOKS
华章科技

作者在Freescale公司从事数字电路设计多年，拥有丰富的设计经验。
首本以专题方式梳理数字电路设计技术的书籍，数字电路设计领域的扛鼎之作。
选图丰富，初学者与中高级读者都能够通过本书完善和提高自己的知识结构。
低功耗、消抖、电磁兼容等内容在一般的书中都鲜有涉及。

电子与嵌入式系统
设计译丛

Springer



The Art of Hardware Architecture
Design Methods and Techniques for Digital Circuits

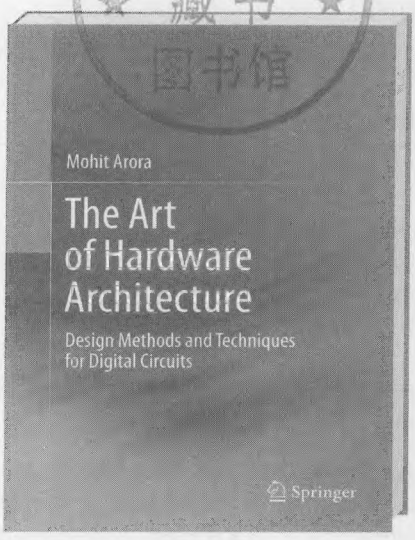
硬件架构的艺术

数字电路的设计方法与技术

［印度］ Mohit Arora 著　李海东 来萍 师谦 等译

机械工业出版社
China Machine Press

电子与嵌入式系统
设计译丛



The Art of Hardware Architecture
Design Methods and Techniques for Digital Circuits

硬件架构的艺术

数字电路的设计方法与技术

[印度] Mohit Arora 著 李海东 来萍 师谦 等译

机械工业出版社
China Machine Press

图书在版编目（CIP）数据

硬件架构的艺术：数字电路的设计方法与技术 /（印）阿罗拉（Arora, M.）著；李海东等译．
—北京：机械工业出版社，2014.2
（电子与嵌入式系统设计译丛）
书名原文：The Art of Hardware Architecture: Design Methods and Techniques for Digital Circuits

ISBN 978-7-111-44939-3

Ⅰ．硬… Ⅱ．①阿… ②李… Ⅲ．数字电路－电路设计 Ⅳ．TN79

中国版本图书馆 CIP 数据核字（2013）第 284096 号

本书版权登记号：图字：01-2013-1816

本书揭示硬件架构的设计艺术，涵盖作者从事芯片设计行业十多年的经验和研究成果。本书共分 9 章，第 1 章介绍亚稳态的概念、量化方法和减少其影响的技术；第 2 章介绍同步设计的时钟技术，并提出可行的时钟方案以及系统复位策略。第 3 章介绍在设计中使用异步时钟或“处理多个时钟”时会出现的问题及解决方法。第 4 章介绍时钟分频器的各个方面和实现方法。第 5 章讲述低功耗设计技术，以减少动态和静态功耗。第 6 章介绍如何把流水线技术应用在处理器的设计中，从而提高性能；第 7 章讨论使用最佳字节顺序的方法；第 8 章阐述去抖动技术，以消除毛刺和噪声。第 9 章介绍电磁干扰的原理、规程、标准和认证，以及电磁干扰的影响因素和减少电磁干扰的方法。

硬件架构的艺术：数字电路的设计方法与技术

（印） 阿罗拉（Arora, M.） 著

出版发行：机械工业出版社（北京市西城区百万庄大街 22 号 邮政编码：100037）
责任编辑：谢晓芳
印　刷：北京市荣盛彩色印刷有限公司
开　本：186mm×240mm 1/16
书　号：ISBN 978-7-111-44939-3
版　次：2014 年 3 月第 1 版第 1 次印刷
印　张：13.5
定　价：59.00 元

凡购本书，如有缺页、倒页、脱页，由本社发行部调换
客服热线：（010）88378991 88361066
购书热线：（010）68326294 88379649 68995259
投稿热线：（010）88379604
读者信箱：hzjsj@hzbook.com

译 者 序

我从2006年开始接触芯片设计行业，在近7年的工作过程中，深刻体会到了这个行业在市场、技术和从业人员规模上急速的扩张与变化所产生的强烈的爆发感。市场热点层出不穷，生产工艺、设计技术和EDA工具的飞速进步，以及国内IC公司如雨后春笋般的涌现和对人才的大量需求，使这个行业在大多数时间处于一种加速的状态中。而作为一名技术人员，在开始的几年里，在加速推力和有限技术积累的阻力这对矛盾中，尤其能体会到其中的激动和迷茫。在edacn（现在已关闭）、eetop和国外一些专业网站上长期交流的日子里，我认识了许多同行的朋友，有国内的，有国外的，大家普遍的共识是专业资料既太多，又太少。太多是指相对于初学者，所关注的主题往往能找到无数的文章论文和技术文档，太少是指对于高级技术人员，真正能将设计提高一个层次的关键技术要点，很难直接遇到。这样的技术要点往往需要资深工程师系统的经验总结，这正是国内IC业界所欠缺的。

在机械工业出版社的张国强编辑的推荐下，我接触到了本书。从头到尾读过这本书后，我的第一感觉就是，实用且有效。作者尽可能用最少的理论基础作铺垫，系统打造出一幢由实用技术组成的大厦，其工程师的思维方式在这里展现无余，一切从解决问题出发，直奔主题，解释怎样做，并给出原理图和代码，以及解决方案。简洁而专注，这就是本书的风格。使用本书可以快速解决设计中可能遇到的问题，也可以很容易地将本书的内容直接应用到设计中。

本书的主要内容涉及时钟和复位、多时钟域设计、时钟分频器、低功耗设计技术、流水线技术、字节顺序、消抖技术和电磁兼容性等方面。绝大部分内容是进行数字设计时必然会接触到的。但也有一些技术在进行某些特殊部分设计时才会涉及，如消抖技术和电磁兼容性。第2章介绍同步设计的时钟技术，并提出了可行的时钟方案，此外也介绍了系统复位策略。第3章介绍多时钟设计的问题和处理方法，几种可能的跨时钟域情况和跨时钟域数据传输方法等。第4章介绍奇数、偶数与小数分频电路的实现和优缺点。第5章介绍数字电路功耗来源，并分别从系统级、体系结构级、寄存器传输级和晶体管级提出一系列降低功耗的方法。第6章介绍流水线的基本原理。第7章说明小端和大端字节顺序的含义，并比较其优缺点和适用领域，以及在进行系统设计时处理使用不同字节顺序IP的方法，此外介绍了字节顺序中性编码规则。第8章介绍典型的开关行为和软硬件消抖技术。第9章介绍电磁干扰的原理、规

程、标准和认证，电磁干扰的影响因素及减少电磁干扰的方法。

本书第1～7章由李海东翻译，第8～9章由来萍、师谦和肖庆中负责翻译。此外，感谢同事李坤在模拟电路方面、曹杰和詹东友在数字电路方面对我的帮助与支持。最重要的，感谢家人对我的理解和支持，如果没有他们的鼓励我无法在节假日坚持每天起早贪黑的翻译工作。

由于译者水平有限，翻译中难免有错误或不妥之处，此外也可能偏向于自己的工作经验，真诚希望各位读者在阅读本书时能将发现的错误及时告知，以便进行更正。

李海东

前　言

我从2000年起开始进入芯片设计行业。工作中涉及的众多研究使我有机会编写相关的技术论文，参加各种技术会议并分享实践经验。在这个过程中，我的出版物得到了许多反馈。常常有读者希望我能将所有的实践经验汇集成一本书。然后这本书问世了。本书重点强调了IP设计者在设计高度优化和可靠的数字电路时所要面对的任务与要掌握的所有技巧。如果恰当使用本书中提到的技术，将大大减少将最初设想转化为实际设计所用的时间，并能保证一次流片成功。

本书读者广泛。本书主要面向半导体行业中的需要深入理解相关主题的芯片设计人员，此外本书也可以作为本科生或研究生的教材。

本书与同类书籍的区别在于本书主要关注芯片设计各领域中所遇到的实际问题，而不仅仅停留在理论概念层面。

本书介绍芯片设计中各方面的顺序如下：从第1章介绍基本概念开始，逐步增加深度到达更高级的主题。如EMC设计技术或复杂的低功耗设计技术，如DVFS（Dynamic Voltage and Frequency Scaling，动态电压频率调整）。

第1章介绍“亚稳态”，可以使读者对其有更为清晰的理解，涉及量化的方法和减少其影响的技术。

第2章围绕设计人员在设计模块或知识产权（IP）时使用的“时钟和复位”提出一系列建议。这些规则独立于任何CAD工具或硅工艺并可应用于任意ASIC设计。

第3章介绍在设计中使用异步时钟或“处理多个时钟”时会出现的问题及解决方法，以使设计能在多时钟域中稳定工作。

第4章介绍“时钟分频器”的各个方面，这是可能需要产生若干个与相位相关的时钟的典型SoC。除了以2为幂的同步分频，该章也介绍了奇数分频（3分频、5分频等），以及非整数分频（1.5分频、2.5分频等）。

第5章介绍“低功耗设计技术”。随着工艺节点变小和应用环境对功耗要求的提高，功耗问题逐渐成为设计的显著约束条件。该章在各种设计抽象级上提出了多种设计方法学和技术，来减少动态和静态功耗。

第6章介绍“流水线”技术。将这种技术应用在处理器的设计中，能提高单位时钟的数

据吞吐率。该章将流水线概念从处理器提高到典型电路中，以提高性能。

第7章介绍“字节顺序”的概念。在设计中各种IP可能有不同的字节顺序，该章介绍使用最优字节顺序的方法。

第8章介绍硬件和软件的“消抖”方法，以除掉从外部输入（通常是某种开关）的有害噪声和毛刺。

第9章深入介绍EMC/EMI的相关问题，将其应用于数字电路的方式，以及为达到“更好的EMC性能”在设计的各抽象级所要遵循的设计规则。

我已经有意将理论部分尽可能精简，主要是为了帮助读者对各专题的理解。各章中所提到的设计准则独立于任何CAD工具或硅工艺，所以设计者可在设计和实施任何片上系统（SoC）项目时使用它们，以得到结构合理且可综合的RTL代码。

本书中有少量的章节包含硬件描述语言（HDL）代码，主要是针对刚刚接触数字电路的初学者。对于已熟练掌握该基础的高级工程师，此部分可以直接跳过。

某些更高级的章节，如第9章，在撰写时经过了几个月的透彻研究，以使其更易于被数字设计人员接受。

我已采取了各种措施以尽可能保持本书的完善性。欢迎广大读者针对这些方面提出反馈或建议。相关意见可以发送至邮箱：mohit. arora@ me. com 或 mohit. arora@ freescale. com。

致谢

我写本书的初衷是想将我多年的工作经验和所进行的众多研究联系起来。然而，如果没有其他人的大力支持和帮助，我是无法完成这样一本书的。

我心爱的妻子Pooja经常在深夜耐心陪伴着我，真心感谢她对我的真诚支持和鼓励。我的大多数写作工作是在周末、夜间、旅途中，以及其他本属于家庭生活的时间进行的。谢谢我的父母，从儿时起就一直支持我坚持完成自己的梦想。

十分感谢Freescale半导体公司的同事Prashant Bhargave耐心阅读了我的初稿，并基于其多年的工作经验提出许多建设性意见。他在本书的编辑、格式调整方面，以及第7章的若干章节中做出了贡献。

特别感谢Freescale半导体公司的Rakesh Pandey和Sudhi Ranjan Proch的早期帮助与对一些章节的反馈。

谢谢Springer的高级编辑Charles Glaser，他为我提供了能将这本书在Springer出版的机会。此外，也对Springer的出版团队中使本书成为一本出色专业读物提供帮助的人表示感谢。

Mohit Arora

目　录

译者序
前　言

第1章　亚稳态的世界 …… 1
1.1　简介 …… 1
1.2　亚稳态理论 …… 1
1.3　亚稳态窗口 …… 3
1.4　计算 MTBF …… 4
1.5　避免亚稳态 …… 5
1.5.1　使用多级同步器 …… 6
1.5.2　使用时钟倍频电路的多级同步器 …… 6
1.6　亚稳态测试电路 …… 7
1.7　同步器的类型 …… 8
1.8　亚稳态/综合性建议 …… 10

第2章　时钟和复位 …… 11
2.1　概述 …… 11
2.2　同步设计 …… 12
2.2.1　避免使用行波计数器 …… 12
2.2.2　门控时钟 …… 12
2.2.3　双边沿或混合边沿时钟 …… 13
2.2.4　用触发器驱动另一个触发器的异步复位端 …… 13
2.3　推荐的设计技术 …… 14
2.3.1　避免在设计中出现组合环路 …… 14
2.3.2　避免数字设计中的延迟链 …… 16
2.3.3　避免使用异步脉冲产生器 …… 16
2.3.4　避免使用锁存器 …… 17
2.3.5　避免使用双沿时钟 …… 20
2.4　时钟方案 …… 22
2.4.1　内部产生的时钟 …… 22
2.4.2　分频时钟 …… 24
2.4.3　行波计数器 …… 25
2.4.4　多路时钟 …… 25
2.4.5　同步时钟使能和门控时钟 …… 26
2.5　门控时钟方法学 …… 28
2.5.1　不含锁存器的门控时钟电路 …… 28
2.5.2　基于锁存器的门控时钟电路 …… 30
2.5.3　门控信号 …… 32
2.5.4　重组数据路径以减少转换传播 …… 32
2.6　复位信号的设计策略 …… 32

2.6.1 用同步复位进行设计…… 33
2.6.2 使用异步复位进行设计…… 36
2.6.3 带异步复位和异步置位的触发器…… 38
2.6.4 移除异步复位的问题…… 39
2.6.5 复位同步器…… 40
2.6.6 过滤复位毛刺…… 41
2.7 控制时钟偏移 …… 42
2.7.1 短路径问题…… 43
2.7.2 时钟偏移和短路径分析…… 43
2.7.3 使时钟偏移最小化…… 45
参考文献 …… 49

第3章 处理多个时钟 …… 50
3.1 介绍 …… 50
3.2 多时钟域 …… 50
3.3 多时钟域设计的难题 …… 51
3.3.1 违背建立时间和保持时间…… 52
3.3.2 亚稳态…… 53
3.4 多时钟设计的处理技术 …… 53
3.4.1 时钟命名法…… 53
3.4.2 分块化设计…… 54
3.4.3 跨时钟域…… 54
3.5 跨时钟域 …… 57
3.5.1 同频零相位差时钟…… 57
3.5.2 同频恒定相位差时钟…… 58
3.5.3 非同频、可变相位差时钟…… 59
3.6 握手信号方法 …… 63
3.6.1 握手信号的要求…… 64
3.6.2 握手信号的缺点…… 64
3.7 使用同步 FIFO 传输数据 …… 65
3.7.1 同步 FIFO 架构 …… 65
3.7.2 同步 FIFO 的工作方式 … 66
3.8 异步 FIFO（或双时钟 FIFO） … 68
3.8.1 避免用二进制计数器实现指针…… 69
3.8.2 使用格雷码取代二进制计数…… 69
3.8.3 用格雷码实现 FIFO 指针…… 72
3.8.4 FIFO 满和 FIFO 空的产生…… 76
3.8.5 双时钟 FIFO 设计 …… 79
参考文献 …… 82

第4章 时钟分频器…… 83
4.1 介绍 …… 83
4.2 同步整数分频器 …… 83
4.3 具有 50% 占空比的奇数整数分频 …… 84
4.4 非整数分频（非 50% 占分比） …… 86
4.4.1 具有非 50% 占空比的 1.5 倍分频 …… 86
4.4.2 4.5 倍分频计数器的实现（非 50% 占空比） …… 87
4.5 *N* 分频的替换方法 …… 88
参考文献 …… 89

第5章 低功耗设计…… 90
5.1 介绍 …… 90
5.2 功耗源 …… 90
5.3 在各设计抽象层次降低功耗 …… 91
5.4 系统级低功耗技术 …… 93
5.4.1 片上系统方法…… 93
5.4.2 硬件/软件划分 …… 93

5.4.3 低功耗软件 …… 95
5.4.4 选择处理器 …… 96
5.5 体系结构级降低功耗技术 …… 97
5.5.1 高级门控时钟 …… 97
5.5.2 动态电压频率调节 …… 99
5.5.3 基于缓存的系统体系结构 …… 100
5.5.4 对数 FFT 体系结构 …… 100
5.5.5 异步（无时钟）设计 … 100
5.5.6 电源门控 …… 102
5.5.7 多阈值电压 …… 105
5.5.8 多电压供电 …… 106
5.5.9 存储器电源门控 …… 106
5.6 在寄存器传输级降低功耗 …… 107
5.6.1 状态机编码和解码 …… 107
5.6.2 二进制数表示法 …… 108
5.6.3 门控时钟基础 …… 109
5.6.4 独热码多路器 …… 111
5.6.5 除掉多余的转换 …… 112
5.6.6 资源共享 …… 114
5.6.7 使用行波计数器来降低功耗 …… 114
5.6.8 总线反转 …… 117
5.6.9 高活跃度网络 …… 118
5.6.10 启用和禁用逻辑云 …… 119
5.7 寄存器级低功耗技术 …… 120
5.7.1 技术水平 …… 120
5.7.2 版图优化 …… 120
5.7.3 衬底偏压 …… 120
5.7.4 减少氧化层厚度 …… 121
5.7.5 多氧化层器件 …… 121
5.7.6 利用定制设计减小电容 …… 121
参考文献 …… 122

第 6 章 流水线的艺术 …… 123
6.1 介绍 …… 123
6.2 影响最大时钟频率的因素 …… 124
6.2.1 时钟偏移 …… 125
6.2.2 时钟抖动 …… 125
6.3 流水线 …… 127
6.4 解释流水线——一个真实的例子 …… 129
6.5 来自于流水线的性能提高 …… 130
6.6 DLX 指令集的实现 …… 133
6.7 流水线对吞吐率的影响 …… 137
6.8 流水线原理 …… 138
6.9 流水线冒险 …… 138
6.9.1 结构冒险 …… 139
6.9.2 数据冒险 …… 140
6.9.3 控制冒险 …… 143
6.9.4 其他风险 …… 144
6.10 ADC 中的流水线——一个例子 …… 145
参考文献 …… 146

第 7 章 处理字节顺序 …… 147
7.1 介绍 …… 147
7.2 定义 …… 147
7.3 小端模式或大端模式：哪个更好 …… 149
7.4 处理字节顺序不匹配的问题 …… 151
7.5 访问 32 位存储器 …… 152
7.6 处理字节顺序不匹配 …… 153
7.6.1 保持数据完整性（数据不变） …… 154
7.6.2 地址不变 …… 156
7.6.3 软件字节交换 …… 158
7.7 字节顺序中性代码 …… 159

7.8 字节顺序中性编码指南…… 159
参考文献 …… 160

第8章 消抖技术 …… 161

8.1 简介…… 161
8.2 开关行为…… 162
8.3 开关种类…… 163
8.4 消抖…… 164
8.4.1 RC 消抖…… 164
8.4.2 硬件消抖电路 …… 168
8.4.3 软件消抖电路 …… 169
8.4.4 消抖指南 …… 171
8.4.5 在多重输入下消抖 …… 172
8.5 现有的解决方案…… 173

第9章 电磁兼容性能设计指南…… 175

9.1 简介…… 175
9.2 定义…… 175
9.3 电磁干扰理论及与电流和频率之关系…… 177
9.4 电磁干扰的规程、标准和认证…… 178
9.5 影响集成电路抗干扰性能的几个因素…… 179
9.5.1 作为噪声源的微控制器 …… 179
9.5.2 影响电磁兼容性的其他因素 …… 180
9.5.3 噪声载体 …… 181
9.6 减少 EMC/EMI 的技术 …… 181
9.6.1 系统级技术 …… 182
9.6.2 板级技术 …… 184
9.6.3 微控制器级技术 …… 193
9.6.4 软件层级技术 …… 196
9.6.5 其他技术 …… 203
9.7 总结…… 204

第1章

亚稳态的世界

1.1 简介

在同步系统中，数据相对于时钟总有固定的关系。当这种关系满足器件的建立和保持时间的要求时，输出端会在特定的传输延迟时间内输出一个有效状态。因为在同步系统中输入信号总是满足触发器的时序要求，所以不会发生亚稳态。但是，在异步系统中，由于数据和时钟的关系不是固定的，因此有时会出现违反建立和保持时间的现象。当违反建立和保持时间时，就会输出介于两个有效状态之间的中间级电平且无法确定停留在中间状态的时间，或者在经过一定的延迟后才能进行正常转换。

本章将帮助读者更清楚地了解有关亚稳态的问题，明白它是如何量化的，以及如何最大限度地减少它的危害。

1.2 亚稳态理论

亚稳态是由于违背了触发器的建立和保持时间而产生的。设计中任何一个触发器都有特定的建立和保持时间，在时钟上升沿前后的这段时间窗口内，数据输入信号必须保持稳定。如果信号在这段时期发生了变化，那么输出将是未知的或者称为“亚稳的”。这种有害状态的传播就叫做亚稳态。触发器的输出会因此而产生毛刺，或者暂时保持在不稳定状态而且需要较长时间才能回到稳定状态。

如图1.1所示，当触发器处在亚稳态时，输出会在高低电平之间波动，这会导致延迟输出转换过程，并超出所规定的时钟到输出的延迟值（t_{co}）。亚稳态输出恢复到稳定状态所需的超出t_{co}的额外时间部分称为稳定时间（t_{MET}）。并非所有不满足建立和保持时间的输入变化都会导致亚稳态输出。触发器是否进入亚稳态和返回稳态所需时间取决于生产器件的工艺技术与外界环境。一般来说，触发器都会在一个或者两个时钟周期内返回稳态。

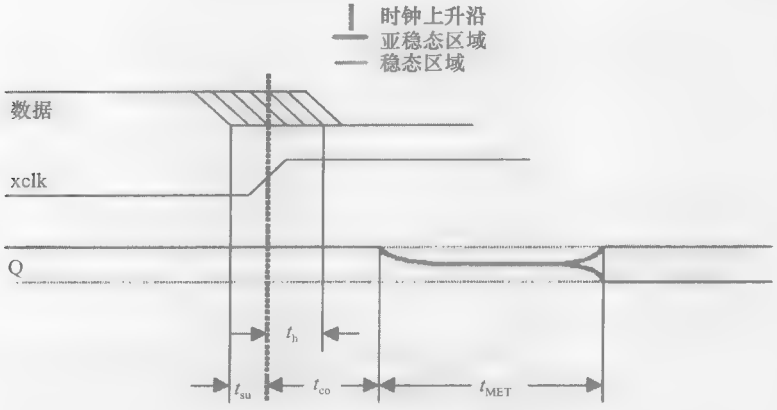


图1.1 亚稳态时序参数

如图1.2所示，触发器的运转类似于在光滑的山上滚动球，山的两边代表两个稳定状态（即高和低），山顶就代表亚稳态。假设球处在一个稳定的状态（即1或0），给球一个足够（满足建立和保持时间要求）的推力（状态转换），使这个球在规定时间内越过山顶到达另一个稳定的状态。

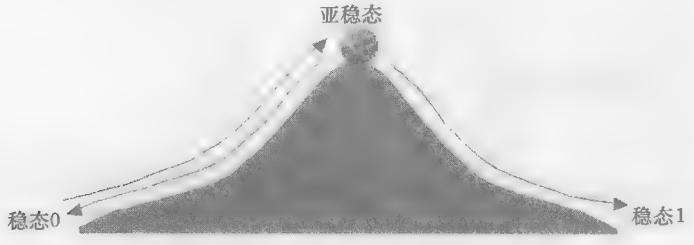


图1.2 触发器的亚稳态行为

然而，如果推力不够（即违反建立和保持时间），这个球就会到达山顶（即输出亚稳态），停留一段时间后再返回到一个稳定的状态（即最终

输出稳定）。这个球也可能会上升一段路程就返回了（即输出可能产生毛刺）。这两种情况都会增加从时钟变化到稳定输出的延迟。

所以，简单地说，当信号在一个时钟域（src_data_out）里变化，在另一个时钟域（dest_date_in）内采样时，就会导致输出变成亚稳态。这就是所谓的同步失败（见图 1.3）。

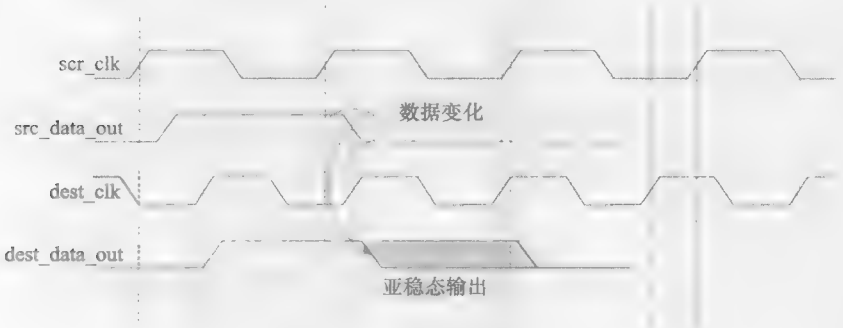


图 1.3 触发器中的亚稳态

1.3 亚稳态窗口

亚稳态窗口（Metastability Window）具有特定的时间长度，在这段时间内输入信号和时钟都应该保持不变。如果它们发生变化，输出就可能变成亚稳态。如图 1.4 所示，建立时间和保持时间共同决定亚稳态窗口的宽度。

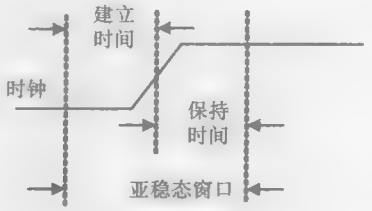


图 1.4 亚稳态窗口

窗口越大，进入亚稳态的概率越高。在大多数情况下，较新的逻辑器件会有更小的亚稳态窗口，也就意味着器件进入亚稳态的概率会更小。

1.4 计算 MTBF

当系统的故障率恒定时，MTBF（Mean/Average Time Between Failures，平均无故障时间）就是故障率的倒数。我们可以从中知道特定触发器发生故障的频率。

对于一个具有给定时钟频率和在该时钟周期内具有均匀概率密度的异步数据信号边沿的单级同步器，亚稳态事件的发生率可以用建立、保持时间窗口和时钟周期的比值乘以信号触发频率来计算。

$$\frac{1}{\text{故障率}} = \text{MTBF}_1 = \frac{e(t_r/\tau)}{Wf_c f_d} \tag{1.1}$$

式中 t_r = 允许超出器件正常传输延迟时间的解析时间

τ = 触发器的亚稳态（解析）时间常数

W = 亚稳态窗口

f_c = 时钟频率

f_d = 异步信号边沿频率

常数 W 和 τ 跟触发器的电气特性有关，会根据工艺技术而改变。所以，相同工艺生产出来的不同器件有着相似的 W 和 τ 值。

如果在不同的解析时间测量和描绘器件的故障率，结果会是一个指数衰减曲线。如图 1.5 所示，当以半对数刻度绘制时，结果就是一条斜率为 d 的直线；因此，在直线上取两个数据点就能够用式 1.2 计算出 τ 的值。

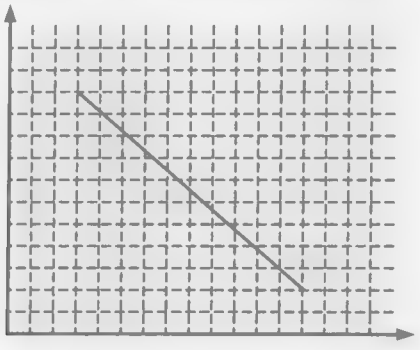


图 1.5 故障率和时间的关系（对数刻度）

$$\tau = \frac{t_{r2} - t_{r1}}{\ln(N1/N2)} \tag{1.2}$$

式中 t_{r1} = 解析时间 1

t_{r2} = 解析时间 2

$N1 = t_{r1}$ 时刻的故障次数

$N2 = t_{r2}$ 时刻的故障次数

基于式 1.1 和式 1.2，两级同步器的 MTBF 可以用式 1.3 来计算。

$$\mathrm{MTBF}_2 = \frac{\mathrm{e}(t_{r1}/\tau)}{Wf_c f_d} \times \mathrm{e}(t_{r2}/\tau) \tag{1.3}$$

式中 t_{r1} = 第一级同步器的解析时间

t_{r2} = 超过正常传输延迟的解析时间

式 1.3 中的第一项计算出了第一级同步器的 MTBF，实质上就是下一级的亚稳态发生率。第二项则根据 t_{r2}（解析时间）的值计算出了亚稳态事件的解析概率。两项结果的乘积就是两级同步器的整个 MTBF。

在给定触发器时钟速率和输入信号变化速率的情况下，如果 MTBF 为 40s，根据式 1.3，用来同步输入的这样的两级触发器的 MBTF 就是 40 × 40 = 26.6min。

1.5 避免亚稳态

如 1.2 节所述，每当违背建立、保持时间时，亚稳态就会出现。在以下条件中，信号可能违背时序要求。

- 输入信号是异步信号。
- 时钟偏移/摆动（上升/下降时间）高于容限值。
- 信号在两个不同频率或者相同频率但是相位和偏移不同的时钟域下跨时钟域工作。
- 组合延迟使触发器的数据输入在亚稳态窗口内发生变化。

亚稳态会引起过多的传输延迟和系统故障，所有的触发器和寄存器都存在亚稳态。虽然亚稳态不能根除，但是可以减小亚稳态发生的概率。

在最简单的情况下，设计人员可以通过确保时钟周期足够长来避免亚稳态，这个时钟周期要大于准稳态的解析时间，也要大于通往下一级触发器的路径上的任何逻辑延迟。虽然这种方法简单，但在大多数现代设计的性能要求下并不实用。另一种避免亚稳态的方法就是使用同步器。

1.5.1 使用多级同步器

避免亚稳态最常见的方法是在跨时钟域的信号上加上一个或者多个同步触发器，如图1.6所示。这种方法用一个完整的时钟周期来解决第一级同步触发器的亚稳态问题（不包括第二级触发器的建立时间）。但是这种方式增加了观察同步逻辑输入的延迟。

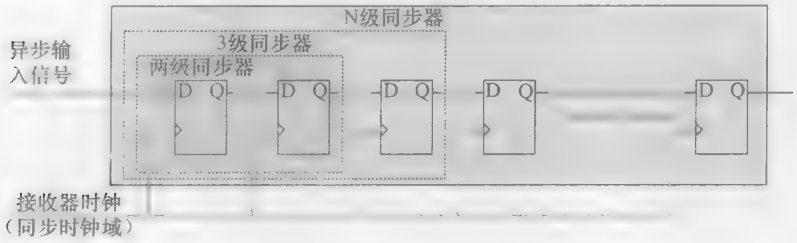


图1.6 N级同步器

1.5.2 使用时钟倍频电路的多级同步器

多级同步器的一个局限就是系统需要花费较长的时间去响应异步输入。解决这个问题的办法就是使用倍频时钟作为两个同步触发器的时钟输入。Altera的FPGA中具有这项称为时钟倍频的技术（见图1.7）。

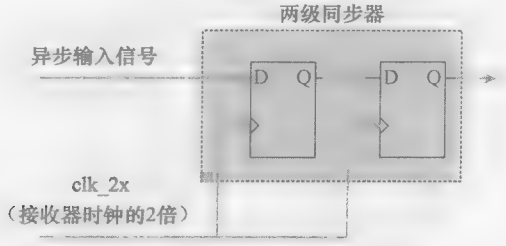


图1.7 带有时钟倍频电路的多级同步器

这种方法不仅能够让系统在一个系统时钟周期内响应一个异步输入，而且改善了MTBF。尽管这种倍频时钟能够降低MTBF，但是这个影响要超过两级触发器引起的偏移量。

两种方法都不能保证同步器阻止亚稳态传播下去，它们仅仅减少了亚稳态发生的概率。

1.6 亚稳态测试电路

每当触发器采样一个异步输入时，触发器输出都可能会产生一个不可预测的延迟，虽然这个概率很低。这不仅会在输入违反建立、保持时间要求时发生，而且在触发器接收新的输入这一小段时序窗口内也会发生。在这些情况下，触发器都会进入亚稳态。

图 1.8 所描述的测试电路可以用来确定触发器的亚稳态特征。图 1.8 给出了有一个异步输入“async_in”的触发器“FF_A”，再由时钟“clk”的上升沿触发。图 1.8 中所示的触发器“FF_B”和“FF_C”都在时钟下降沿触发，这样做是为了捕捉 FF_A 的亚稳态事件。

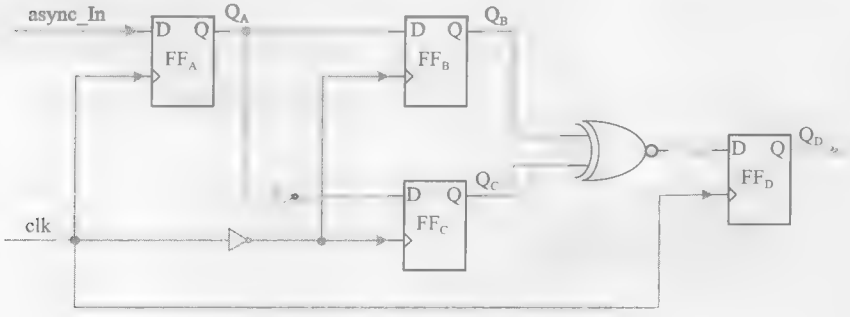


图 1.8 亚稳态测试电路

当两个互补的信号分别传递到“FF_B”和“FF_C”的输入时，无论“FF_A”什么时候出现亚稳态事件，异或非（XNOR）门的输出都会变为高电平。在触发器“FF_D”输出端捕捉到高电平就表明已经检测到亚稳态事件。

测试电路中所有节点的时序如图 1.9 所示。

因为解析触发器（“FF_B”和“FF_C”）由时钟下降沿触发，所以可以通过改变时钟高电平的时间（Δt）来控制所要求的稳定时间。稳定时间 t_{met} 由下式确定

$$t_{met} = \Delta t - t_{ACN} \tag{1.4}$$

式中，t_{ACN} 是最小时钟周期，等于 t_{CQ}（FF_A 的时钟到输出延迟）+ 解析触发器（FF_B 或 FF_C）的建立时间 t_{su}。

减少解析时间或者稳定时间的方法之一就是给建立/保持时间处的数据集中加入抖动。

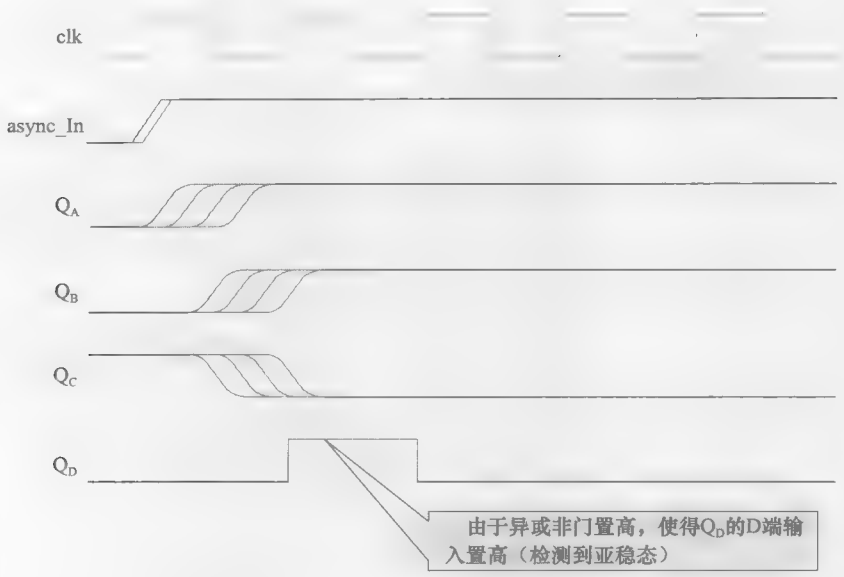


图 1.9 测试电路中的时序

1.7 同步器的类型

根据式 1.1，一个异步输入电路的 MTBF 和用于从亚稳态恢复的时间呈指数关系。用同步器构成的时间缓冲器可以帮助从亚稳态中恢复。

要注意，一个异步信号不应该被两个或者多个同步器所同步（这样做会存在多级同步器输出产生不同信号的风险）。本节介绍两级同步器的两种不同的模式：模式 A 和模式 B。

模式 A 是一个标准的电路，当异步输入信号比时钟周期大得多时最有效（见图 1.10）。

注意，即使异步输入在建立时间区间之外稳定，它仍然需要由时钟驱动产生两个周期的延迟，否则 FF1 可能进入亚稳态。

如果亚稳态在不到一个时钟周期内就解析了，FF2 就会有稳定的输入，否则就需要更深层的级联，如图 1.6 所示。

但是，模式 A 不能用于异步输入信号的宽度小于时钟周期的情况。在这种情况下要采用模式 B，如图 1.11 所示。

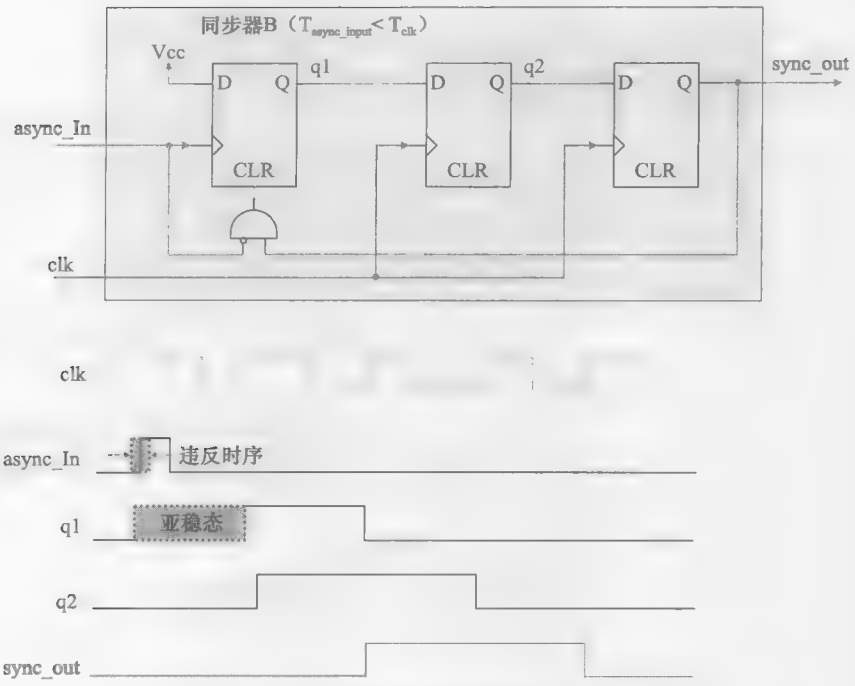


图 1.10 两级同步器的模式 A

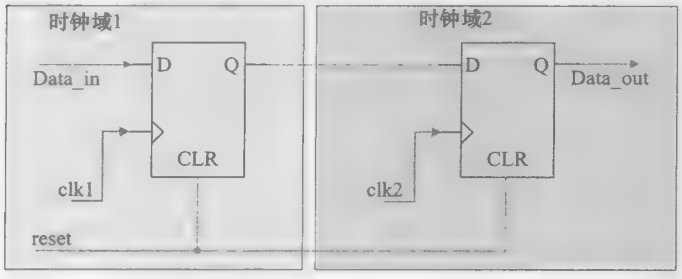


图 1.11 两级同步器的模式 B

注意，对于同步器的模式 B，第一级触发器的输入 D 与 V_{CC} 相连，同时时钟信号是异步输入信号。另外两个触发器直接由系统时钟（clk）控制。一个短脉冲让 q1 变成高电平，这个高电平在经过两个时钟（clk）沿后从 sync_out 输出。

我们总结了如下经验规则

1）当信号必须跨时钟域工作时，要采用同步器。

2）如果 clk1 < clk2，在时钟域 2（见图 1.11）里用同步器的模式 A，否则就采用同步器的模式 B。

1.8 亚稳态/综合性建议

在两个相互异步系统的交界面亚稳态是不可避免的。但是下面的几点建议可以明显减小亚稳态发生的概率。

(a) 采用同步器。

(b) 采用响应更快的触发器（缩短亚稳态窗口 T_W）。

(c) 使用亚稳态硬化触发器（专为高宽带设计并且减少为时钟域输入电路而优化的采样时间）。

(d) 如图1.6所示，使用级联触发器（两个或者多个）作为同步器。如果一个触发器的亚稳态失败概率为 P，那么 N 个触发器的亚稳态失败率就是 P^N。

(e) 减少采样速率。

(f) 避免使用 dV/dt 低的输入信号。

第2章
时钟和复位

2.1 概述

目前设计专用集成电路（ASIC）的成本每年都在增加。除了一次性工程（Non-Recuring Engineering，NRE）成本和掩膜成本，由于专用集成电路的复杂程度的提高，开发费用也有所增多。同时，由于ASIC的设计复杂性使得改善成本也越来越高。为了克服重新设计的风险，高NRE成本和减少上市时间的延迟，保证设计的芯片能在第一次流片后就可以工作变得越来越重要。

本章涵盖了设计者在设计模块或者知识产权（Intelletual Property，IP）时所要用到的一些建议。这些建议独立于任何CAD工具或者硅工艺，并能应用于任何ASIC设计，同时有助于设计者用结构良好的和可综合的RTL代码来计划与运行一个成功的片上系统（System on Chip，SoC）。

当前的设计逐渐向系统级集成（System Level Integration，SLI）转变，即将多个复杂功能模块和多种存储器集成在单块电路上，这对集成水平提出了一系列新的要求。这些建议主要针对模块设计和集成在片上系统中的存储器接口。然而，这里给出的建议跟系统级集成的要求完全一致，这将会极大地减轻集成工作，并且确保这些独立模块可以在其他系统里轻松地复用。

可以以这些建议为基础来制成清单，用于每次设计制造加工之前的检查。

2.2 同步设计

在同步设计中，由单个主时钟和单个主置位/复位信号驱动设计中所有的时序器件。

经验表明对 ASIC 的时域控制最安全的方法就是同步设计。

2.2.1 避免使用行波计数器

用触发器来驱动其他触发器的时钟输入端，一般会存在问题。由于第一个触发器时钟到 q 的延迟而使第二个触发器的时钟输入产生偏移，而且不能在每个时钟边沿都激活。用这种方式连接两个以上的触发器就会形成如图 2.1 所示的行波计数器。注意，由于使用了更多的触发器，会使延迟累积增加，所以不推荐使用这种方式。关于行波计数器的更多细节将在 5.6.7 节中进行介绍。

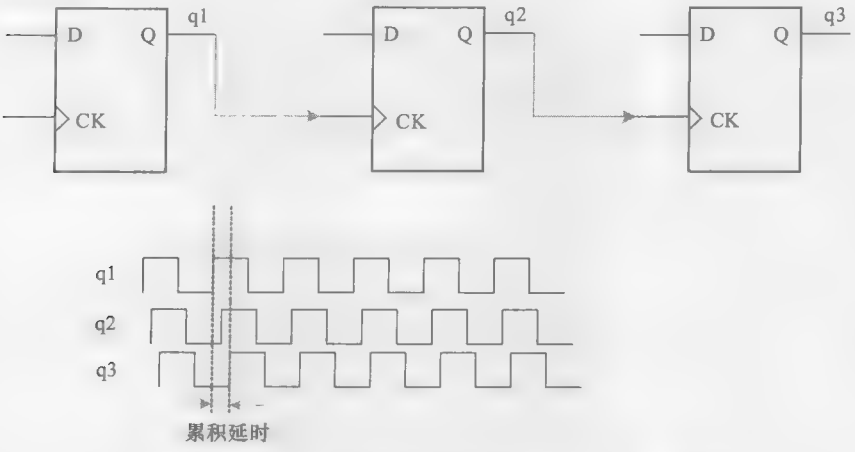


图 2.1 触发器驱动其他触发器的时钟输入端（行波计数器）

2.2.2 门控时钟

时钟线上的门控单元会导致时钟偏移，并会引入尖峰脉冲作用于触发器。在时钟线上存在多路复用器时，这个问题尤为严重，如图 2.2 所示。

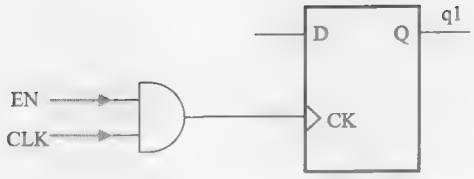


图 2.2 门控时钟线

含门控时钟的设计在仿真时可能工作很正常，但是进行综合时就会出现问题。

2.2.3 双边沿或混合边沿时钟

如图 2.3 所示，两个触发器由两个相位相反的时钟信号控制。这会为使用同步复位和使用插入扫描链这样的测试方法带来麻烦，同时也会增加确定关键信号的路径的难度。

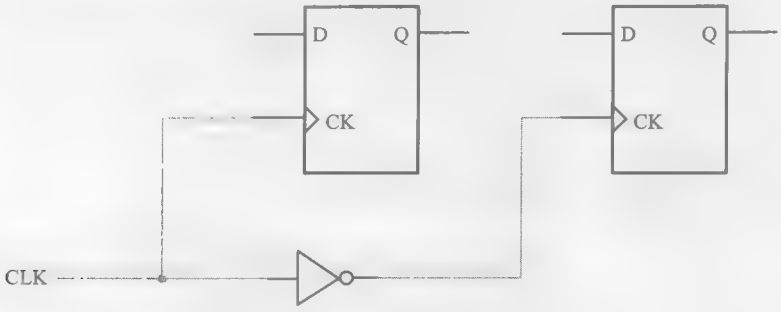


图 2.3 双边沿时钟

2.2.4 用触发器驱动另一个触发器的异步复位端

在图 2.4 中，第二级触发器的输出不仅仅只受时钟边沿的影响，这违反了同步设计原理。此外，该电路还包含第二级触发器时钟和复位之间的潜在竞争条件。

随后的章节介绍的一些方法可以避免使用上述不推荐的电路。

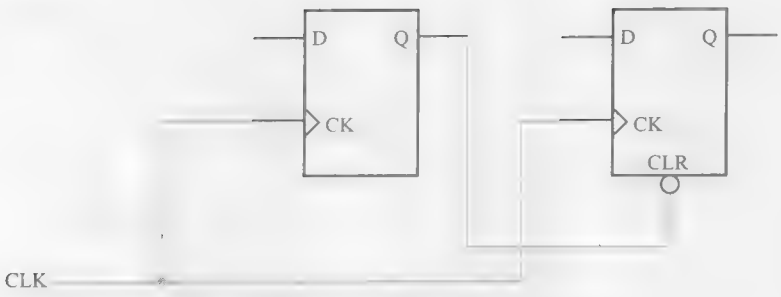


图2.4 触发器驱动另一触发器的异步复位端

2.3 推荐的设计技术

在使用HDL代码进行设计时，理解综合工具是如何对不同的HDL编码风格和结果进行解释是很重要的。从硬件角度思考作为一种特别的设计风格（或相当于编码风格）很重要，这会影响到设计门数和时序性能。本节将讨论一些基本的设计技术，以得确保得到优化的设计结果，同时避免不可靠和不稳定。

2.3.1 避免在设计中出现组合环路

组合环路是数字设计中导致不稳定和不可靠的最常见因素。在同步设计中，所有反馈回路都应包含寄存器。组合环路建立了不含寄存器的直接反馈回路，这违背了同步设计原理。

在HDL语言中，当信号经过若干组合always[⊖]块产生自身时，或者算术表达式左边的内容也出现在右边时，就会形成组合环路。组合环路对设计是一种冒险，在遇见组合环路时综合工具总是会报错，它是不可综合的。

组合环路的产生可以从图2.5的气泡图中理解。每个气泡表示一个组合always块，进入其中的箭头表示该always块要用到的信号，从气泡中出去的箭头表示该块的输出信号。很明显信号“a”通过信号“d”依赖于自身产生，因此形成了组合环路。

⊖ 为了简单起见，本书指代的任何HDL语言以Verilog为例。

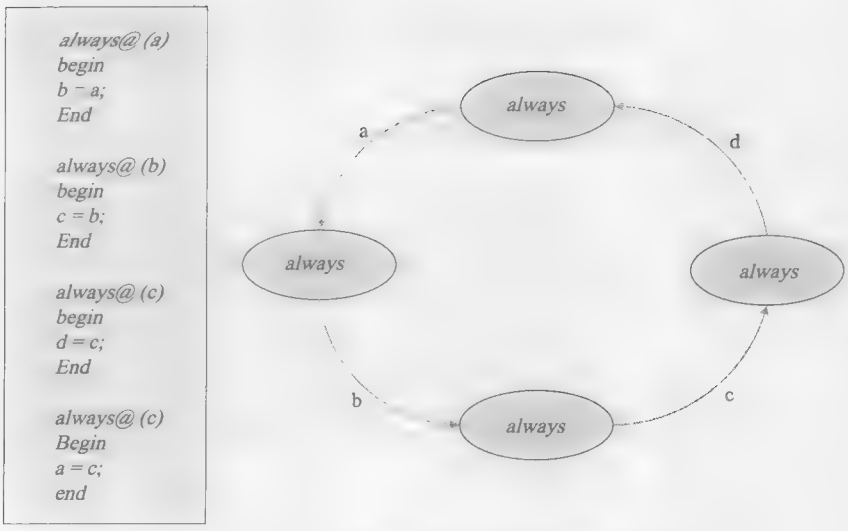


图 2.5 组合环路的例子和气泡图

代码和气泡图在下面列出[28]：

为了移除组合环路，必须改变其中一个信号的生成方式，以消除信号对彼此的依赖性。解决这个问题的简单方法是在组合环路中引入一个触发器或寄存器，以将直接通路打断。

图 2.6 展示了另一个例子，这里寄存器的输出通过组合逻辑直接控制同一个寄存器的异步输入端。

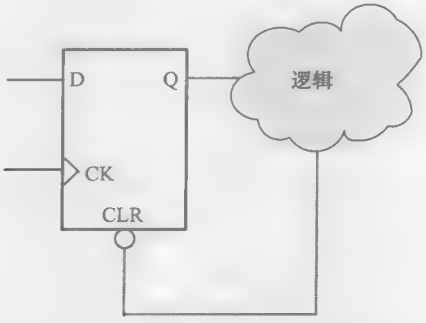


图 2.6 通过异步控制端口的组合环路

组合环路是固有的高风险设计结构。组合环路的行为与该环路中所有逻辑的传播延迟相关。因为传播延迟会根据不同条件而改变，所以组合环路的行为也可能变化。在许多设计工具中，组合环路都会导致无休止的循

环运算。在设计流程中使用到的各种工具可能会以不同的方式将组合环路打断，以与原始设计意图不一致的方式对其进行处理。

2.3.2 避免数字设计中的延迟链

在用两个或多个带有单扇入和单扇出的连续节点产生延迟时，就会形成延迟链。通常将反向器链在一起以增加延迟。延迟链通常出现在异步设计中，有时用来解决由其他组合逻辑导致的竞争条件。在 FPGA 和 ASIC 中，都会因为布局布线产生延迟。延迟链能导致各种各样的设计问题，包括增加设计对操作环境的敏感性，降低设计的可靠度，以及增加将设计移植到不同器件结构上的难度。避免使用延时链，需要在设计中用同步技术取代异步技术。

2.3.3 避免使用异步脉冲产生器

设计通常要求基于某些事件产生脉冲。设计人员有时会使用延迟链产生单个脉冲（脉冲产生器）或一系列脉冲（多振子）。有两种常用方法产生脉冲。这些技术是纯异步的，应尽可能避免使用。

- 将同一个触发信号接到两输入与门或或门的两个输入端，但是对其中一个输入端的信号取反或加入延迟链。脉冲的宽度取决于直接接到门输入端的信号和经过延迟后接到门输入端的信号的相对延迟。这与组合逻辑中由输入端变化引起毛刺的原理是一样的。这种技术通过使用延迟链人为地增加了毛刺宽度。
- 寄存器输出经过延迟链后驱动同一个寄存器的异步复位端。该寄存器本质上经过一个确定的延迟后对自身异步复位。

异步产生的脉冲宽度常常为综合和布局布线软件带来难题。实际的脉冲宽度只有在布局布线之后，在布线和传播延时已知时才能确定。所以很难在创建 HDL 时就确定可靠的延迟值。脉冲宽度可能不适用于所有 PVT 环境，并且转移到不同技术节点时脉冲宽度也会变化。除此之外，因为静态时序分析不能用来验证脉冲宽度，所以验证工作会变得困难。

多振子利用“毛刺产生器”的原理创建脉冲，此外组合环路将该电路转变为振荡器[25]。

因为涉及对脉冲数量的控制，所以产生多脉冲的结构会比脉冲产生器引起更多的问题。此外在使用该结构产生多脉冲时，也增加了设计的频率。

推荐使用的同步脉冲产生器如图 2.7 所示。

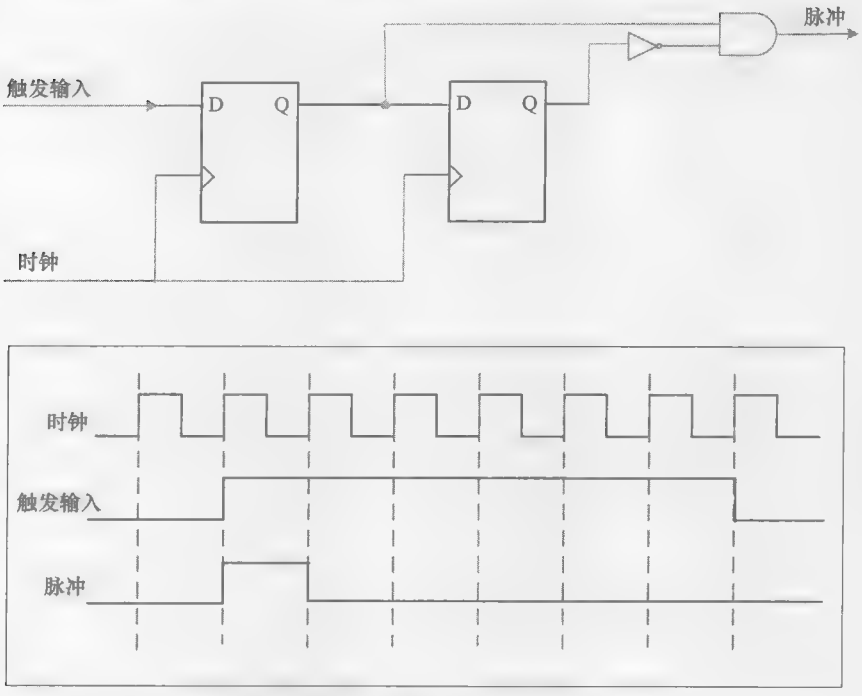


图 2.7 触发器输入起始处的同步脉冲产生器

在上面的同步脉冲产生器中，脉冲宽度总是与时钟周期宽度相等。该脉冲产生器是可预测的，可以用时序分析进行验证，并易于移植到其他体系结构中，同时独立于具体的工艺。

与图 2.7 类似，图 2.8 是在触发输入末端产生脉冲的脉冲产生器。

2.3.4 避免使用锁存器

在数字设计中，锁存器用来在新的值到来前保持原来信号的值。避免在所有可能的位置使用锁存器，而应用触发器代替。

如图 2.9 所示，如果 X 和 Y 信号都变为高，因为锁存器由电平触发，所以将它们同时开启会使电路产生振荡。

锁存器会为设计增加各种各样的问题。虽然锁存器是与寄存器相似的存储器件，但是它们有根本的区别。锁存器是连通模式的，即在数据输入和输出之间存在直接通路。输入端的毛刺能传递到输出端。

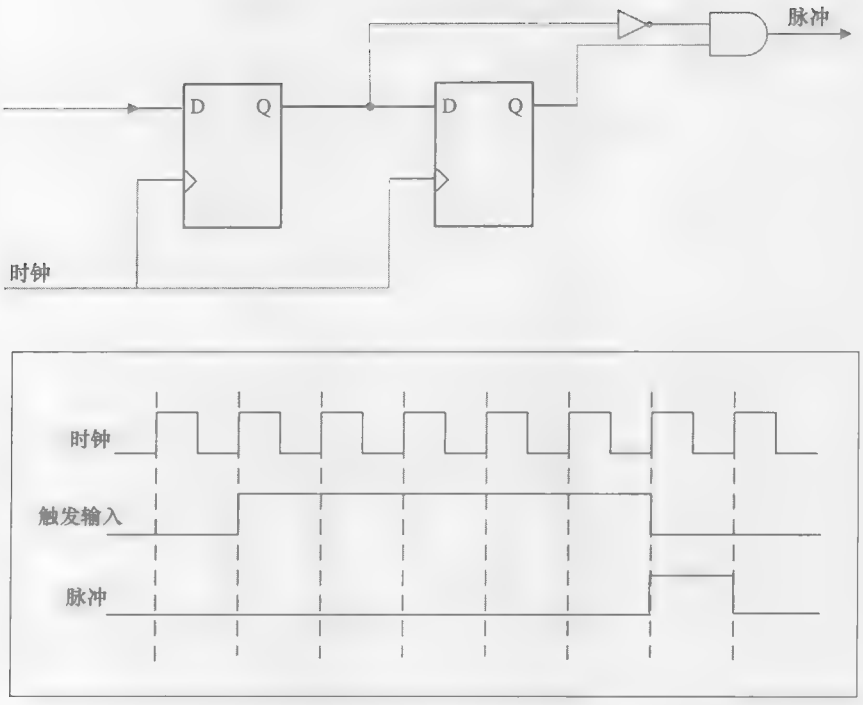


图 2.8 触发器输入末端的同步脉冲产生器

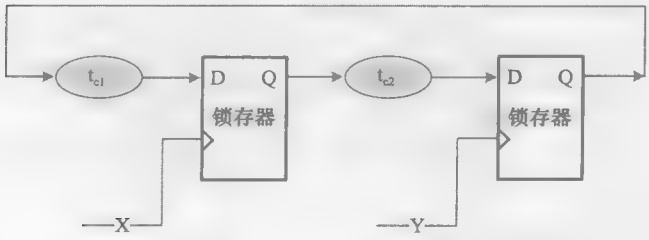


图 2.9 锁存器的竞争条件

静态时序分析器通常会作出与锁存器透明有关的错误假设，并要么发现通过数据输入端口的伪路径要么将真正的关键路径丢失。锁存器本身的时序也是模糊的。例如，在分析含 D 锁存器的电路时，工具无法确定你是想在时钟前沿还是在时钟后沿将数据传输到输出端。大多数情况下，只有原始设计者才知道设计的全部意图，这也就意味着其他设计者很难移植同样的设计或复用代码。

锁存器常常使电路不可测。大多数可测性设计（Design For Test，

DFT）工具和自动测试程序生成器（Automatic Test Program Generator，ATPG）都不能很好地处理锁存器。

锁存器给 FPGA 设计带来了不同的挑战，因为 FPGA 是寄存器密集型的；所以使用锁存器的设计会占用更多逻辑，并且比使用寄存器的设计性能更低。

综合工具有时会推断出设计意图之外的锁存器。这样的锁存器往往源自不完整的“if”或“else”语句。忽略“if”或“case”表达中最终的“else”分句也会形成锁存器。图 2.10 给出了一个类似的例子。

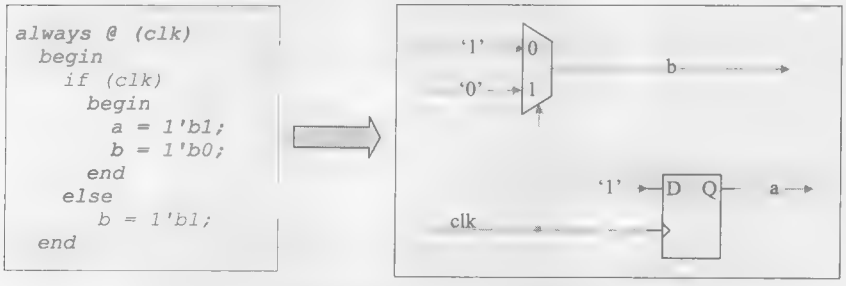


图 2.10 由不完整的“if else”描述推断的锁存器

如图 2.10 所示，把‘b’综合成直连的组合逻辑，‘a’将推断为锁存器。

推断锁存器的一般规则是，如果一个变量未能在 always 语句的所有可能执行条件下赋值（例如，当一个变量没有在 if 语句的所有分支中赋值时），就会形成锁存器。

某些 FPGA 架构不支持锁存器。当综合包含锁存器的设计时，综合工具会创建组合反馈通路来取代锁存器（如图 2.11 所示）。

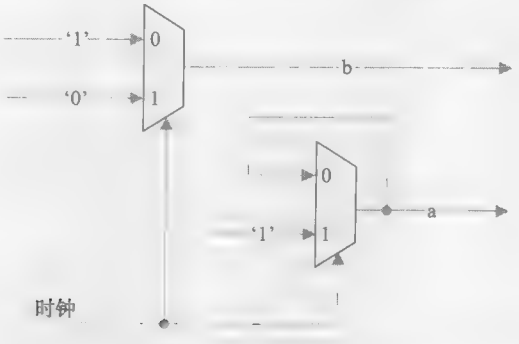


图 2.11 由不完整的“if else”描述实现的组合环路

虽然图2.11中的组合反馈通路有锁存数据的能力，但是造成了比锁存器更多的问题，因为这种电路结构可能违背建立时间和保持时间的要求，而且很难发现，相比之下，由于锁存器由电平触发，因此并没有建立时间和保持时间的问题。

注意

设计中不应包含任何组合反馈环路。应该用触发器、锁存器或完整的RTL枚举条件来取代它。

总而言之，这并不意味着锁存器永远不能存在，我们将在后面看到使用锁存器挪用周期或借用时间来满足关键路径的要求，这是一件多么有趣的事。

2.3.5 避免使用双沿时钟

使用双沿时钟是指在时钟的上升沿和下降沿都传输数据。这种改变使得数据传输在给定的时钟速率下能达到双倍的吞吐率。

双沿输出级时钟是一种增加设计最大可能输出速度的有效方法，但是它违反了同步电路的原理并将导致一系列问题。

图2.12给出了一个由双沿时钟触发的电路。

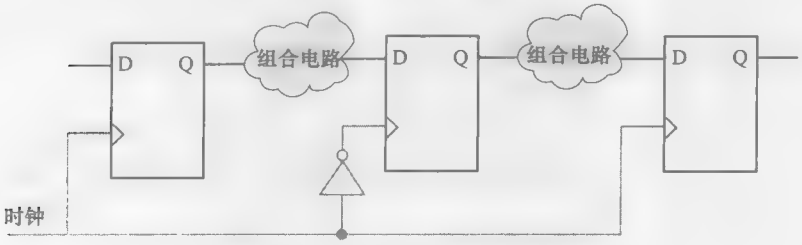


图2.12 由双边沿时钟驱动的逻辑

使用双沿时钟时遇到的一些问题如下：

- 不对称的时钟占空比会导致违背建立和保持时间。
- 很难确定关键信号的路径。
- 很难使用像插入扫描链这样的设计方法学，因为它要求所有寄存器使用同样的时钟边沿。如果要在双沿电路中插入扫描链，那么必须在时钟线上插入多路复用器以保证在测试模式下使用单一时钟。

图 2.13 中是一种使用单个时钟的通常等效流水线逻辑。注意，该同步电路要求的时钟频率是图 2.12 的两倍。

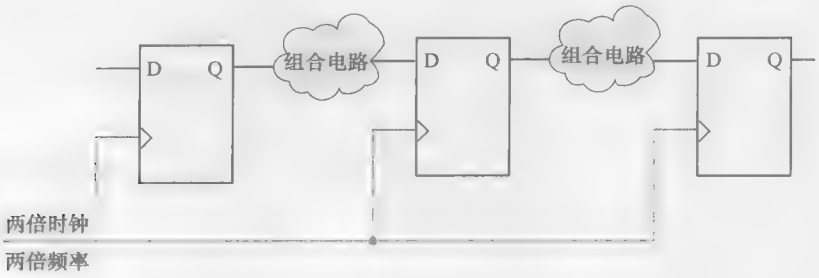


图 2.13 使用单沿时钟的逻辑

图 2.14 展示了由时钟驱动的单沿数据传输和多沿数据传输过程。

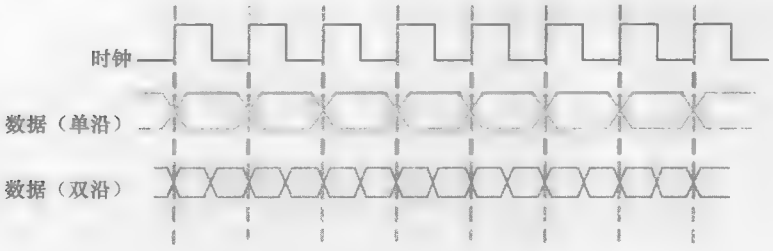


图 2.14 单/双沿数据传输

绿色和蓝色信号表示数据；六边形形状是表示信号的惯用方式，在任何给定时刻信号值可能是 1 或 0。

在图 2.12 中，时钟非对称占空比可能导致违背建立和保持时间，而且扫描路径也不能很容易地通过触发器。

这并不意味着永远不能使用双沿时钟。在对性能/速度要求很高的情况下，并且无法承受使用等效同步电路在 DFT 和验证方面所带来的额外开支时，也可以使用双沿时钟。

双沿时钟的优点

PC 世界中永恒不变的信条是提升性能。这也就意味着大多数接口在不断地改进，以使用更高的时钟来提高吞吐率。PC 世界中的许多较新技术已经超越了单一提高时钟频率的方式。它们改变了接口或总线传输信号的整体方式，使数据在每个时钟周期内不止传输一次，而是两次或更多。

使用双沿时钟比使用两倍时钟频率的等效同步电路有其他的好处。无

论程度如何，接口设计人员通常都会提高系统的时钟频率。然而，过高的时钟频率会给许多接口带来问题。最常见的问题就是接口本身的电气特性。随着时钟频率和时序的增加干扰会变得越来越敏感，使得接口电路必须制作得更精细以处理更高的时钟频率，这就增加了成本。

另外一个使用双沿时钟的优势是降低功耗，因为时钟频率减半，所以系统的功耗只有等效同步电路的一半。

总之，除非等效同步电路无法满足所期望的性能时，系统设计人员才应该使用双沿时钟。

2.4 时钟方案

2.4.1 内部产生的时钟

设计者应该尽可能避免在内部产生时钟，因为如果操作不当，它会导致设计功能和时序问题。

组合逻辑搭建的时钟产生器会引入毛刺，使功能出现问题，此外由组合逻辑所导致的延迟也会导致时序方面的问题。在同步设计中，数据输入端的毛刺不会引起任何问题，因为数据是在时钟边沿处捕获的，所以可以将毛刺自动滤掉。然而，如果毛刺或尖峰脉冲出现在时钟输入端（或者寄存器的异步输入端）就会产生明显的影响。

窄毛刺会违背寄存器的最小脉冲宽度要求。在毛刺到达时钟输入端时，如果寄存器的数据输入变化，会违背建立和保持时间。即使设计没有违背时序要求，寄存器也可能输出意料之外的值，使整个设计功能出现风险，这可能在设计中的任何位置出现。

图2.15展示了使用组合逻辑在同步计数器上产生时钟的效果。在时序图中可以看到，由于时钟沿处的毛刺，计数器在所示的时钟周期上递增了两次。

由于时钟毛刺的作用，计数器增加了额外的计数值，这样就可能导致功能出现问题。

注意

这时为了方便起见，假设计数器在毛刺处的数据没有违背建立和保持时间方面的要求。

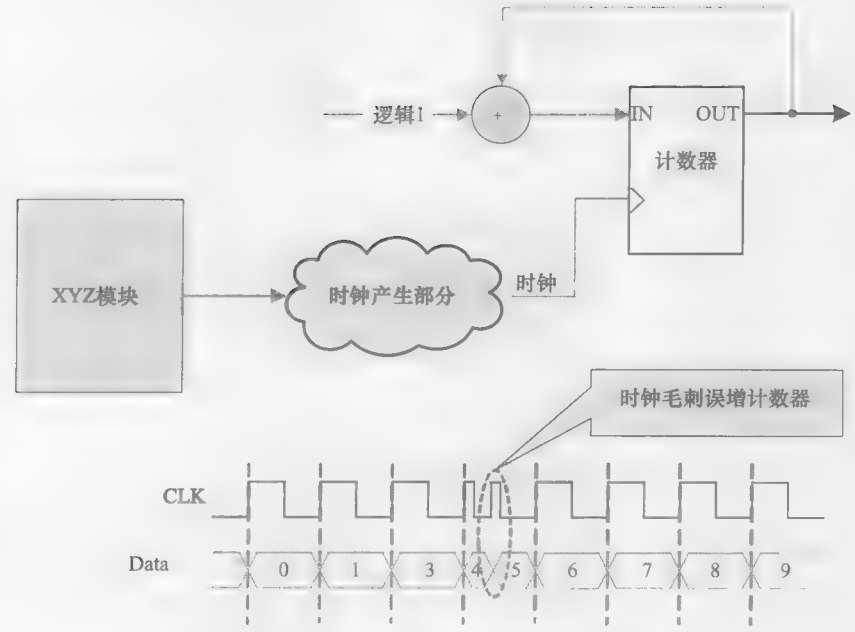


图 2.15 使用组合逻辑作为时钟的计数器例子

解决上述问题的一个简单方法是在组合逻辑的输出端增加一个寄存器，用寄存器的输出作为后面的时钟信号。这个寄存器可以保证在寄存器的数据输入端阻止组合逻辑所产生的毛刺（见图 2.16）。

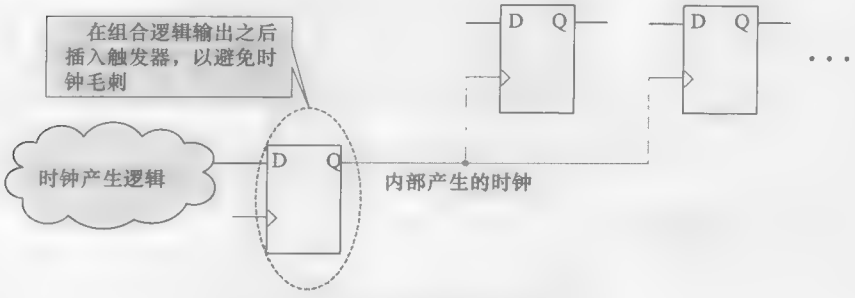


图 2.16 推荐的时钟产生技术

用来产生内部时钟的组合逻辑也会增加时钟线上的延迟。在某些情况下，时钟线上的逻辑延迟会导致时钟偏移比两个寄存器之间的数据路径延迟更大。如果时钟偏移大于数据延迟，就会违背寄存器的时序要求，设计的功能也不会正确。

图2.17给出了一个由时钟路径延迟导致在输入“IN”端违背建立时间的例子。

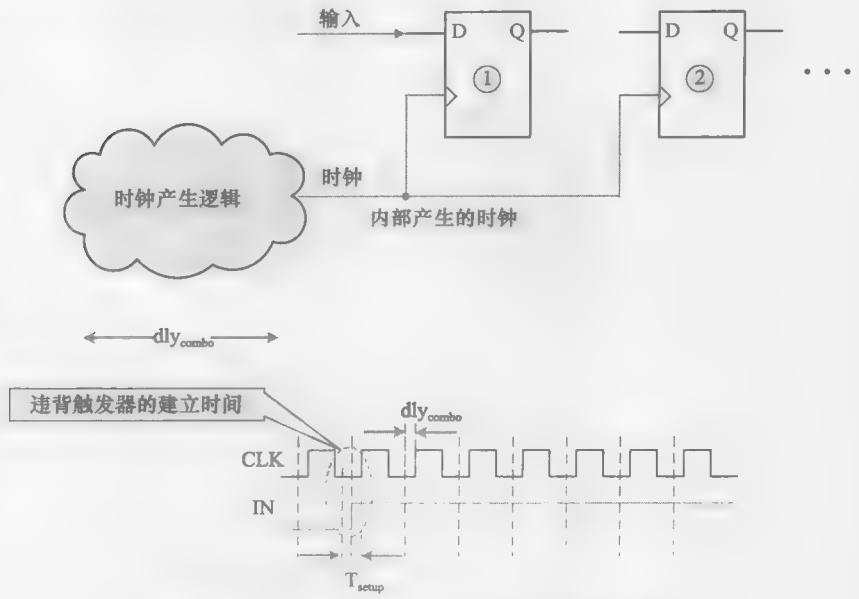


图2.17 由于时钟路径偏移导致违背建立时间

注意

为简单起见，数据通路延迟假设为0。

一种减少时钟偏移的方法是将产生的时钟放到SoC中高扇出且低偏移值的时钟树上。使用低偏移值时钟树有助于减少信号整体的时钟偏移。

2.4.2 分频时钟

许多设计需要来自于主时钟的分频时钟。在设计中要保证大多数时钟来自于PLL。使用PLL能避免由异步时钟分频逻辑引起的许多问题。在对主时钟进行分频时，应该始终使用同步计数器或状态机。

此外，设计应该保证总是由寄存器直接产生分频时钟信号。永远不要对计数器或状态机的输出进行解码，然后产生时钟信号；这种实现方式常常会导致毛刺和尖峰脉冲。

2.4.3 行波计数器

ASIC 设计人员常常使用行波计数器对时钟进行幂为 2 的分频，与其他同步计数方式相比，行波计数器使用的门数更少。行波计数器使用级联寄存器，即每个寄存器的输出引脚连接到下一级寄存器的时钟引脚上（见图 2.18）。

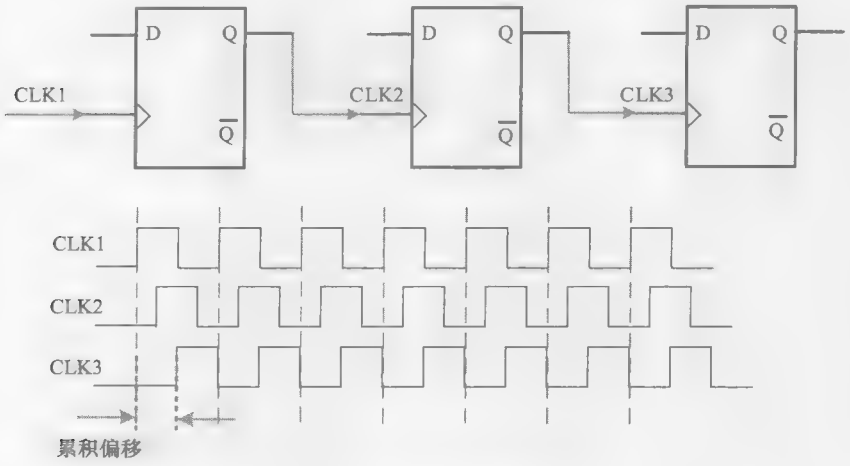


图 2.18 行波计数器的级联效应

因为计数器在各阶段创建行波时钟，所以这种级联会导致问题。这些行波时钟会对 STA 和综合工具带来麻烦。所以应该尝试避免使用这种结构以减少验证的工作量。

尽管使用行波计数器存在各种挑战和问题，但是在功耗较高的系统中很适合使用这种计数器，因为这样能大量降低由逻辑或 SoC 所引起的峰值功耗。

注意

在少数的情况下数字设计师可以考虑使用这种技术，但需严格控制。

参考第 5 章以理解使用行波计数器来节省功耗的更多细节。

2.4.4 多路时钟

时钟多路器用于使同一个逻辑功能具有不同的时钟。某些类型的多路

逻辑选择图 2.19 中所示的时钟源。

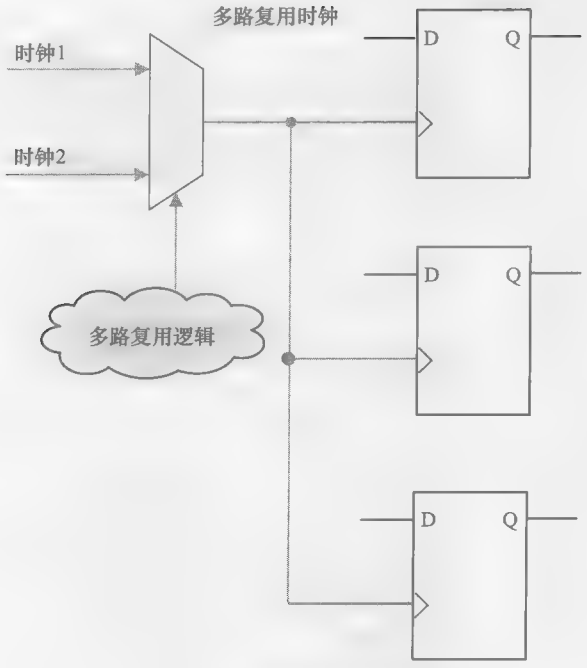


图 2.19 多路逻辑和时钟源

例如，需要处理多个频率标准的通信应用常常使用多个时钟。

虽然在时钟信号上引入多路逻辑会引入前面章节中所提及的问题，但是在不同的应用中对多路时钟的要求差别很大。

如果能满足下面的标准，时钟多路操作就是可接受的：

- 在初始化配置后，时钟多路逻辑就不再改变。
- 在测试时，设计会绕过功能时钟多路逻辑而选择普通时钟。
- 在时钟切换时，寄存器始终处于复位状态。
- 在时钟切换时产生的短暂错误响应没有负面影响。

如果设计中时钟切换很频繁，并且不在复位时切换，设计也不能容忍芯片中出现短暂的错误响应，就必须使用同步设计以确保寄存器没有违背时序，时钟信号上不出现毛刺同时没有竞争条件或其他麻烦。

2.4.5 同步时钟使能和门控时钟

门控时钟使用使能信号开关时钟，实现对某些门控电路的控制。如

图 2.20所示，在时钟关闭时，相应的时钟域就会关闭，其功能会无效。

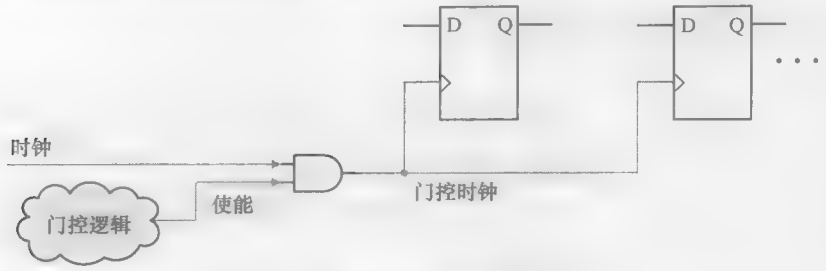


图 2.20 门控时钟

门控时钟是减少功耗的有力手段。在时钟被门控关闭后，该时钟网络和其中的寄存器都会停止翻转，因此它们的功耗减少。然而，由于门控时钟并不是同步设计方案的一部分，因此会显著增加设计时间和验证的工作量。门控时钟会增加时钟偏移并对毛刺敏感，所以能导致设计失败。

通过使用同步的时钟使能，可以纯同步方式关闭时钟域。然而，在使用同步时钟使能方案时，时钟树一直保持翻转而且每个触发器的内部电路保持活跃（虽然触发器输出值没有改变），这样无法降低功耗。同步时钟使能技术如图 2.21 所示。

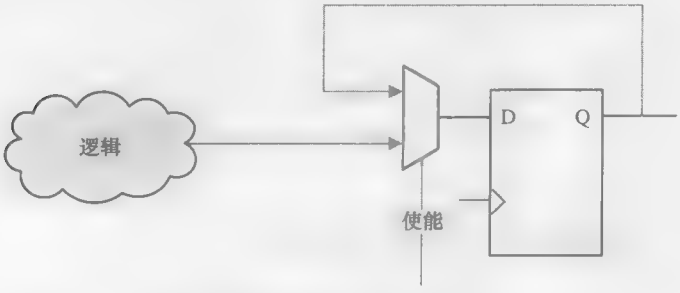


图 2.21 同步时钟使能

由于时钟网络仍然在不停翻转，因此同步时钟使能的时钟方案不能像门控时钟那样从源头减少功耗，但是通过使某些寄存器失效，可以实现与门控时钟同样的功能。如图 2.21 所示，根据使能信号，每个触发器数据输入端前面的多路器要么载入新的数据，要么复制寄存器的输出。

下一节将会重点介绍高效门控时钟方法学，以在对功耗要求严格的设计中使用。

2.5 门控时钟方法学

在传统的同步设计风格中，系统时钟连接到每个寄存器的时钟端。这使得功耗主要由三个部分组成。

1）在每个时钟沿变化的组合逻辑所产生的功耗（由于触发器驱动这些组合逻辑）。

2）由触发器产生的功耗（即使在触发器的输入和内部状态未变化，该功耗仍然存在）。

3）设计中时钟树产生的功耗。

对时钟路径进行门控能大幅降低触发器的功耗。门控时钟可以存在于时钟树的根部、末端，或两者之间的任何位置。

由于时钟树几乎消耗了整个芯片功耗的50%，因此最好始终在根部产生或关闭时钟，以使整个时钟树都关闭，而不是沿时钟树在末端才关闭时钟。

图2.22是一个带门控时钟的三位计数器。

除了把门控时钟器件插入时钟网络中，该电路与传统的实现方式是相同的，这样只有在INC输入为高电平时才由时钟驱动触发器。当INC输入为低电平时，触发器无时钟输入，因此保持原来的值。这样就节省了触发器前面的3个多路器，这3个多路器按图2.21所示的方式在由同步时钟使能实现门控时插入。当实现大的寄存器块时，这会节省大量的面积。

2.5.1 不含锁存器的门控时钟电路

不含锁存器的门控时钟使用一个简单的“与”门或“或”门实现（取决于触发器使用哪个边沿），如图2.23所示。

为了避免过早截断时钟脉冲或误产生多个时钟脉冲（或时钟上的毛刺），正确的操作强制要求使能信号从时钟活跃沿（上升沿）起到时钟不活跃沿（下降沿）止一直保持常量。

图2.24说明了未满足上述要求而使产生的时钟过早被截断的情况。

这种限制使得在基于单时钟触发器的设计中不适合使用不含锁存器的门控时钟。

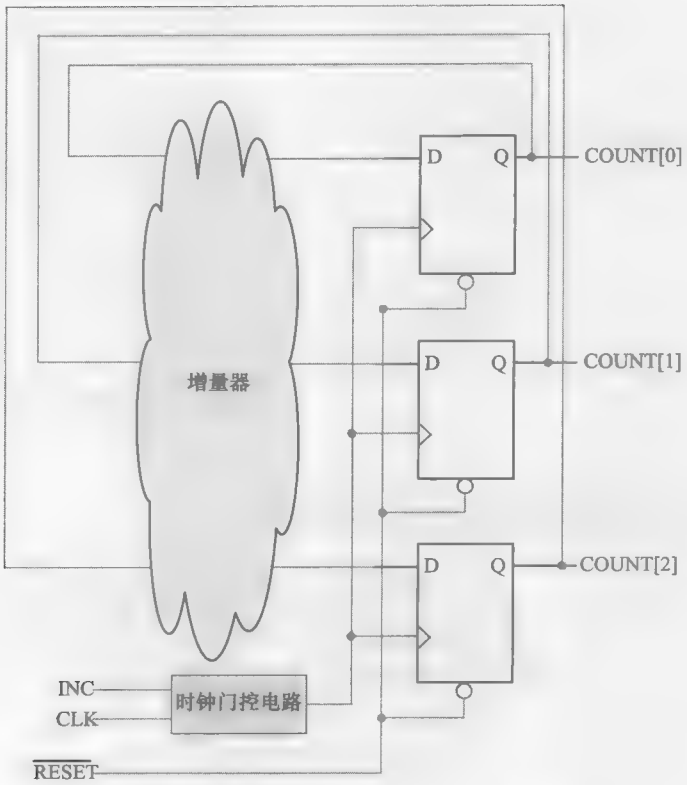


图 2.22 含时钟门控的三位计数器

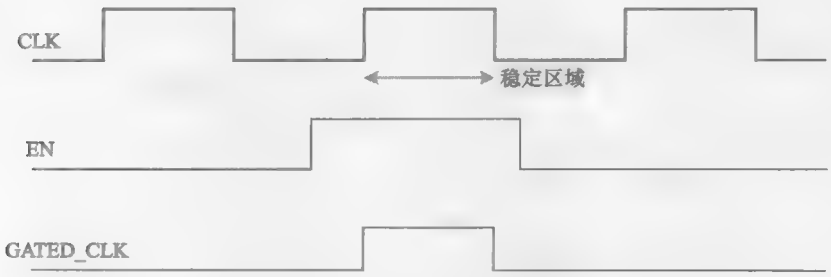


图 2.23 不含锁存器的时钟门控电路

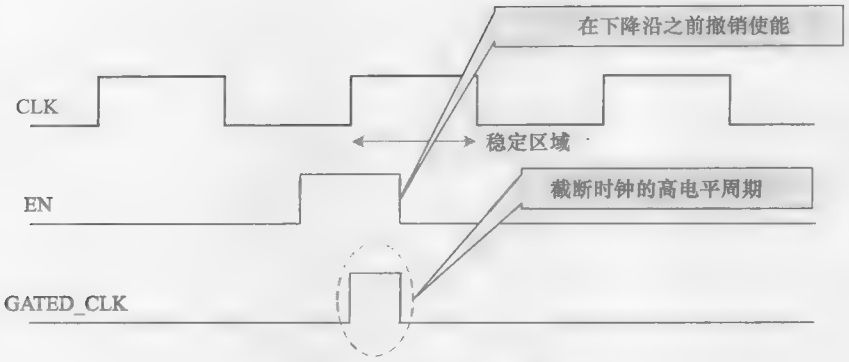


图 2.24 生成时钟过早地终止

2.5.2 基于锁存器的门控时钟电路

基于锁存器的门控时钟风格向设计中加入了一个电平敏感的锁存器，以在时钟活跃沿和不活跃沿之间保持使能信号不变，这样就无须依靠门控电路本身来满足这一要求了，如图 2.25所示。

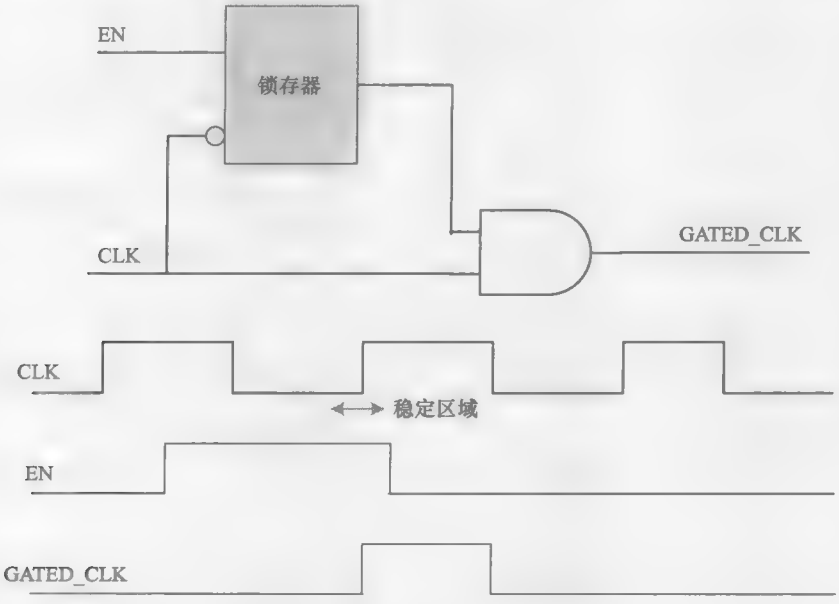


图 2.25 基于锁存器的时钟门控电路

由于锁存器能捕捉到使能信号并使它保持到产生完整的时钟脉冲，因此使能信号只需要在时钟上升沿附近保持稳定即可。

使用这种技术，每次只需要改变门的一个输入端来打开或关闭时钟，就能保证电路的输出不含任何毛刺或尖峰脉冲了。

注意

使用与门控制在上升沿有效的时钟。对于在下降沿有效的时钟，使用或门进行控制，并用正沿触发的锁存器寄存使能信号。

在使用这一技术时，必须特别注意时钟的占空比以及产生使能信号逻辑的延迟，因为使能信号必须在半时钟周期时产生。在产生使能信号的逻辑极为复杂或者时钟占空比失衡时，就会产生问题。与其他产生门控时钟的方式引入的问题相比，关注占空比和逻辑延迟的工作量是可以接受的。

为了保证较高的生产缺陷覆盖率，有必要保证在使用扫描方法学时门控时钟电路是完全可控和可观察的。加入控制信号，使得设计中所有触发器不管使能值是多少都能被时钟驱动，这样就能使扫描链按正常方式移动扫描数据了。

该信号可以在锁存器前先与使能信号进行一次或操作，并可将该信号连接到指示进入扫描测试中的测试模式信号上，或连接到只在扫描移位时有效的扫描使能信号上。

修改后的电路如图 2.26 所示。大多数 ASIC 生产商都提供“门控时钟单元”作为标准单元库的一部分。

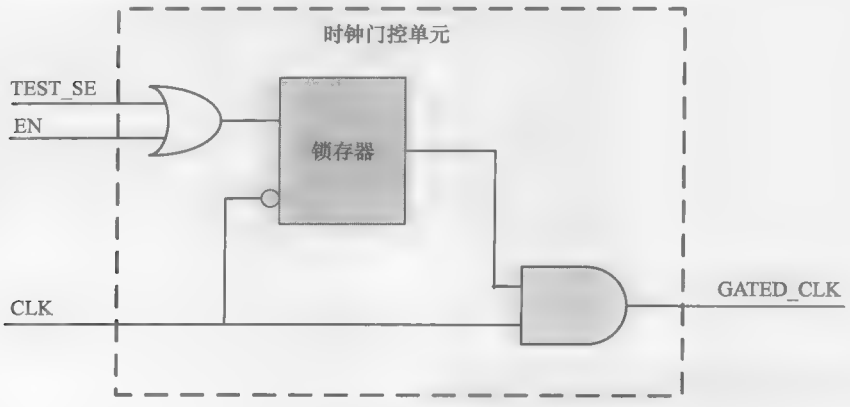


图 2.26 标准时钟门控单元

2.5.3 门控信号

对设计的特殊部分使用门控信号能提高能效。与门控时钟的概念相似，门控信号减少了与时钟无关信号的变化。最常见的例子是解码器使能。

在地址解码机制中，地址信号会在所有目标模块解码器中变化。解码器输入端的切换行为会导致大量的门翻转。可以用使能或选择信号阻止切换行为的传播，但是可能会使逻辑变成稍为复杂（见图2.27）。

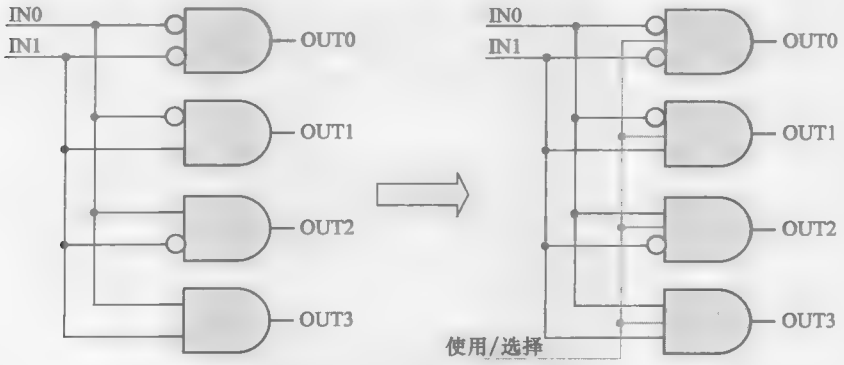


图2.27 带使能端的解码器

2.5.4 重组数据路径以减少转换传播

某些数据路径中的元件，如解码器或比较器，以及“毛刺的”逻辑会显著增加功耗。由较迟到达的信号或偏移所引起的毛刺，可以穿过其他数据通路器件和逻辑到达寄存器。因为信号变化引起了逻辑电平的翻转，所以这种传播消耗了更多的能量。为了减少这种损耗，设计者需要重新编写HDL代码以尽可能地缩短传播路径的长度。图2.28举例说明了两种优先级多路器的实现方式，这里“毛刺的”和“稳定的”条件有不同的排序方式。

2.6 复位信号的设计策略

在为ASIC选择复位策略之前必须考虑许多设计方面的问题，如使用

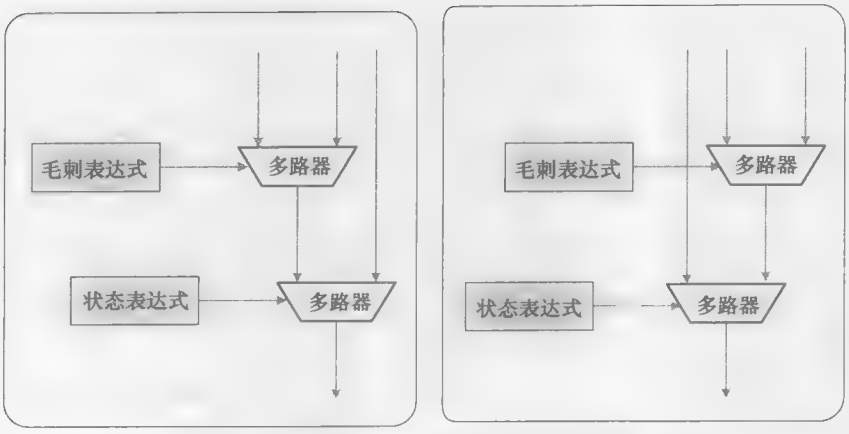


图 2.28 数据通路重排序以减少切换传播

同步复位还是使用异步复位，是否每一个触发器都要接收到复位信号等。

复位最基本的目的是使 SoC 进入一个能进行稳定操作的确定状态。这可以避免 SoC 在上电后进行入随机状态而死机。一旦 SoC 生产出来，是否需要对 SoC 使用复位就由系统、SoC 的应用环境以及 SoC 的自身的设计方式来决定。好的设计指南会在系统没有明确要求的情况下为 SoC 中的每个触发器都提供复位信号。在某些情况下，当流水线的寄存器（移位寄存器触发器）在高速应用中使用时，应该去掉某些寄存器的复位信号以使设计达到更高性能。

设计可以选择使用异步复位或同步复位，或者同时使用两者。两种复位方式各都有各自明显的优点和缺点，在实际的设计中任何一种方法可以有效使用。设计者必须选择最适合于设计本身的复位方式。

2.6.1 用同步复位进行设计

同步复位的复位信号只有在时钟的有效沿到来时才能影响或者复位触发器的状态。在某些仿真器中，根据电路的逻辑，可能会阻止复位信号到达触发器中。这只是仿真中出现的现象，并不存在于真实的硬件中。

由于复位树的高扇出，复位相对于时钟周期可能是一个“迟到的信号”。即使复位信号经过了复位缓冲树的缓冲，也要尽可能减少其到达本地逻辑前穿过的逻辑数量。

图 2.29 展示了带有同步复位的可加载触发器的 RTL 代码。图 2.30 为对应的硬件实现。

```
module load_syn_ff ( clk, in, out, load, rst_n);
  input clk, in, load, rst_n;
  output out;

  always @(posedge clk)
    if (!rst_n)
      out <= 1'b0; // sync reset
    else if (load)
      out <= in; // sync

endmodule
```

图 2.29 同步复位可加载触发器的 Verilog RTL 代码

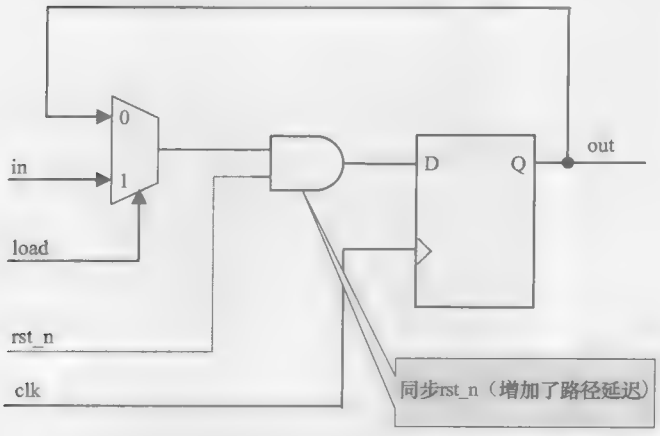


图 2.30 带同步复位的可加载触发器（硬件实现）

使用同步复位会出现的一个问题是综合工具无法分辨复位信号和其他数据信号。综合工具可能产生一种图 2.31 所示的电路结构。

图 2.31 所示的电路在功能上与图 2.30 中的完全一样，区别仅仅在于复位与门在多路器之外。现在，思考在门级仿真时会出现什么现象。当“rst_n”信号为低时，能使多路器的两个输入强制为 0，然而如果“load”是未知状态（X）并且多路器模型是悲观的，这时触发器就会停在未知态（X），而不会复位。注意，这只是仿真过程中出现的问题。实际电路将会正常工作，触发器也会复位到 0 值。

综合工作经常会提供编译指令，以告知综合工具指定的信号是同步复位信号（或置位信号）。综合工具会将该信号“拉”得尽可能接近触发器，以避免初始化问题的发生。

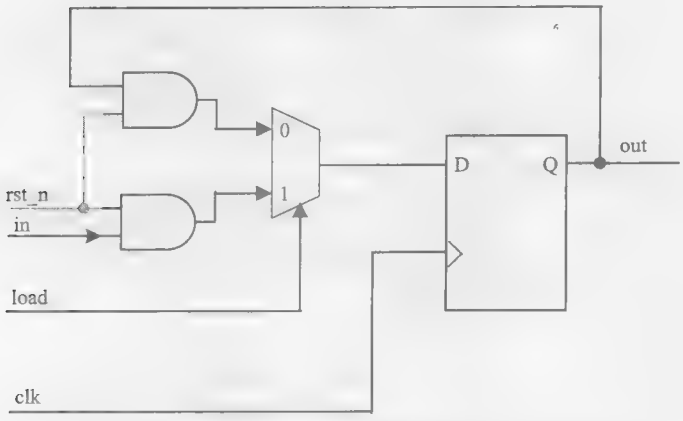


图 2.31 同步复位可加载触发器的另一种实现方式

推荐的方式是在项目开始时把这些指令加入 RTL 代码中以避免重新综合影响到项目进度。

2.6.1.1 使用同步复位的优点

1）同步复位一般能确保电路是 100% 同步的。

2）同步复位会综合为更小的触发器，特别在该复位信号被触发器的输入逻辑门控时。

3）同步复位确保复位只发生在有效时钟沿。时钟可以作为过滤掉复位毛刺的手段。

4）在一些设计中，复位必须由一组内部条件产生。推荐在这样的设计中使用同步复位信号，这样可以将时钟之间的复位毛刺过滤掉。

2.6.1.2 使用同步复位的缺点

并不是所有 ASIC 库都带有内置的同步复位触发器。因为同步复位信号只是一个数据信号，所以很容易把复位逻辑综合到触发器自身之外（如图 2.30 和图 2.31 所示）。

1）同步复位可能需要一个脉冲展宽器，以保证复位信号能出现在时钟有效沿处。在进行多时钟设计时，这是必须要考虑的一个重要问题。可以使用一个小计数器来保证具有指定周期数的复位脉冲宽度。

2）如果复位由 SoC 的组合逻辑产生或复位必定经过多级组合逻辑，就会存在潜在的问题。在仿真过程中，根据复位的产生方式或在功能模块上的使用方式，复位信号可能标记为 X。问题并不是得到了一个什么样的

复位信号，而是该复位信号是否可以容易地被外部引脚控制。

3）就其本质而言，同步复位需要时钟以复位电路。在出于节省功耗的目的而使用门控时钟时，就可能出现问题。在复位信号发出时，时钟可能关闭。在这种情况下只能使用异步复位，并在时钟恢复前移除复位信号。

如果 ASIC/FPGA 有内部三态总线，就迫切需要时钟来产生复位动作。为了阻止芯片上电时内部三态总线出现竞争，芯片应当有图 2.32 所示的异步上电复位。

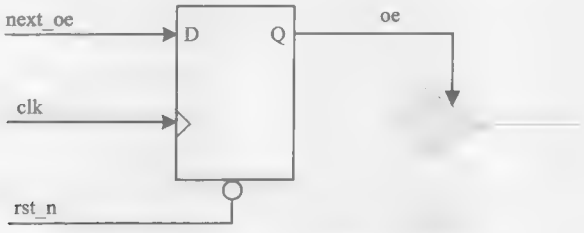


图 2.32 用于输出使能的异步复位

可以使用同步复位信号；但是也必须使用复位信号直接撤销三态使能（见图 2.33）。这种同步技术的优点是能简化复位 - 高阻这一路径的时序分析。

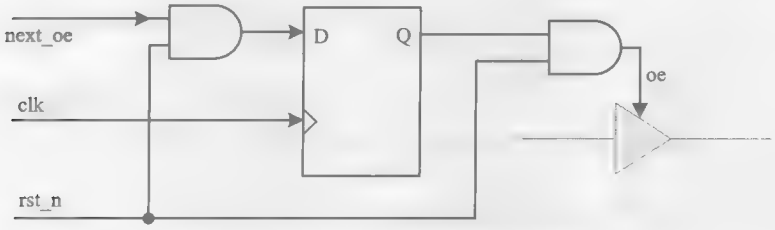


图 2.33 用于输出使能的同步复位

2.6.2 使用异步复位进行设计

异步复位触发器在设计时加入了一个复位引脚。通过低电平有效的复位（在设计中经常使用），当该信号使触发器的复位端变为逻辑低电平时，触发器进入复位状态。

图 2.34 是带有异步复位的可加载触发器的 RTL 代码。图 2.35 是对应

的硬件实现图。

```
module load_asyn_ff ( clk, in, out, load, rst_n);
   input clk, in, load, rst_n;
   output out;

   always @(posedge clk or nededge rst_n)
     if (!rst_n)
       out <= 1'b0; // sync reset
     else if (load)
       out <= in; // sync

endmodule
```

图 2.34 异步复位可加载触发器的 Verilog RTL 代码

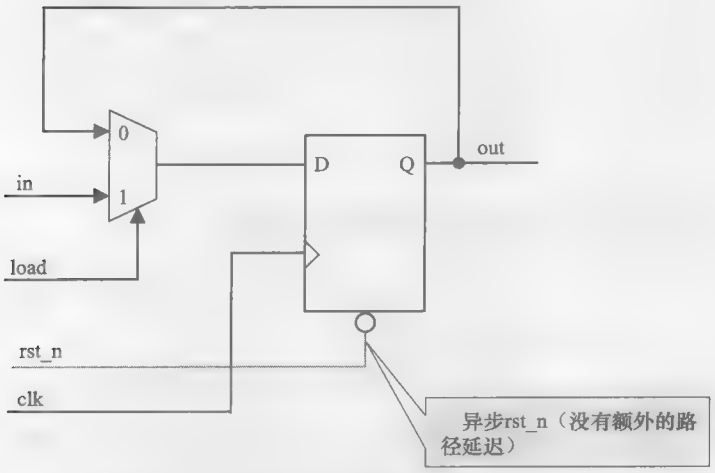


图 2.35 异步复位可加载触发器（硬件实现）

2.6.2.1 使用异步复位的优点

1）使用异步复位的最大好处在于只要生产方提供的库中有带有异步复位的触发器，就能保证数据路径上是干净的。对于数据路径时序已经很紧的设计，无法承担由于加入同步复位带来的额外单元门和额外线路延迟。而使用异步复位，设计人员就能保证没有任何复位信号加在数据路径上（见图 2.35）。

2）异步复位最明显的优势电路是不管有没有时钟都能复位。综合工具能自动推断出异步复位而不必加入任何综合参数。

2.6.2.2 使用异步复位的缺点

1）在DFT时，如果异步复位信号不能直接被I/O引脚驱动，就必须将异步复位线路与复位驱动器断开以保证DFT扫描和测试的正确。

2）异步复位最大的问题是不管产生或撤销复位信号，它们都是一个异步过程。产生复位信号不存在问题，但是撤销时就出现了问题。如果异步复位在触发器时钟有效沿附近释放，触发器的输出就会进入亚稳态，因此SoC的复位状态就会丢失。

3）异步复位的另一个问题与其源头有关，即由板级或系统复位所产生的噪声或毛刺引发的伪复位。需要设计毛刺过滤器来消除复位电路上毛刺的影响。如果在系统中这真是一个问题，那么设计人员就应该考虑使用同步复位的方案。

4）对于同步和异步复位，复位树都应是时间可控的，以确保复位能在一个时钟周期内释放。必须在设计版图后分析复位树的时序以确保满足时序要求。解决这个问题的一种方法是使用分布式复位同步触发器。

2.6.3 带异步复位和异步置位的触发器

大多数设计并没有既包含异步置位又包含异步复位的触发器，但是有时需要这样的触发器。

图2.36是异步置位/复位触发器的Verilog RTL代码。

```
module dff_set_reset ( clk, in, out, rst_n, set_n);
  input clk, in, rst_n, set_n;
  output out;

always @(posedge clk or nededge rst_n or negedge set_n)
  if (!rst_n)
    out <= 1'b0; // async reset
  else if (!set_n)
    out <= 1'b1; // async set
  else
    out <= in;
endmodule
```

图2.36 可异步置位/复位触发器的Verilog RTL代码

综合工具能正确推断出带有异步置位/复位的触发器，但是在仿真时并不是这样。仿真的问题是只在置位、复位或时钟信号的有效沿才进入always块。

如果复位信号首先有效，然后置位信号有效，之后复位信号无效，触发器应该先进入复位状态，然后进入置位状态（见图 2.37 中的时序图）。

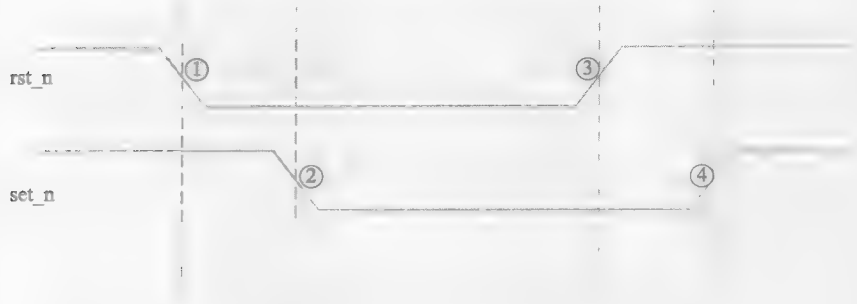


图 2.37 任意异步置位/复位条件下的时序波形

由于以上所有输入都是异步的，因此只要一移除置位复位就应该有效，但是在 Verilog 中并不是这样，因为在下一个时钟上升沿到来前，没有其他方式能触发 always 块。always 块仅对于图 2.37 所示的 1 和 2 事件才触发，并跳过 3 和 4 事件。

在某些很少见的设计中，允许复位和置位同时发出，然后复位先撤销，修复这种仿真问题的方式是将自校验码附着在正确的编译器指令中对触发器建模，并在这种情况发生时将触发器输出强制为正确的值。建议最好尽可能避免使用同时带有异步置位和异步复位的触发器。

图 2.38 所示的代码是使仿真正确，并且能保证综合前和综合后仿真结果一致的修改方案。

```
// Add Compiler specific directive to
// ignore the following block during synthesis
always @(rst_n or set_n)
if (rst_n && !set_n) force q = 1;
else release q;
// End the compiler directive here
endmodule
```

图 2.38 带异步置位/复位触发器的仿真模型

2.6.4 移除异步复位的问题

移除系统中的异步复位会使芯片进入不稳定的未知状态。在释放复位时必须要特别注意避免这种情况的发生。在使用同步复位时，复位信号的

前沿和尾沿必须远离时钟的有效边沿。

如图2.39所示，在移除异步复位信号时，有两个潜在的问题。

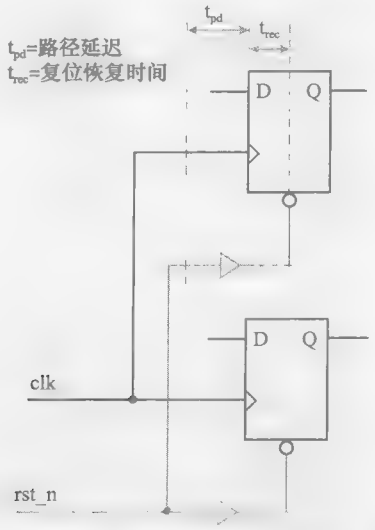


图2.39 消除异步复位的恢复时间问题

1）违背复位恢复时间。复位恢复时间指复位撤销后和时钟再一次置高之间的时间。违背复位恢复时间会使寄存器数据输出端出现数据完整性或亚稳态问题。

2）复位移除会在不同时序元件的不同时钟周期内发生。当复位移除与时钟上升沿异步时，在复位信号和/或时钟信号上不同的传播延迟会导致某些寄存器或触发器提前退出复位状态。

2.6.5 复位同步器

在2.6.4节中提到解决异步复位移除问题的方法是使用复位同步器。对于使用异步复位信号的电路，这是确保正确移除复位最常使用的方法。如果没有复位同步器，即使在仿真时复位能正常工作，最终系统中的异步复位信号仍然是无效的。

图2.40中设计的复位同步逻辑最好用于异步复位和同步复位。

用外部异步复位信号来复位一对触发器，这对触发器异步地依次驱动主复位信号通过复位缓冲树，再到达设计中的其他触发器。然后整个设计异步复位。

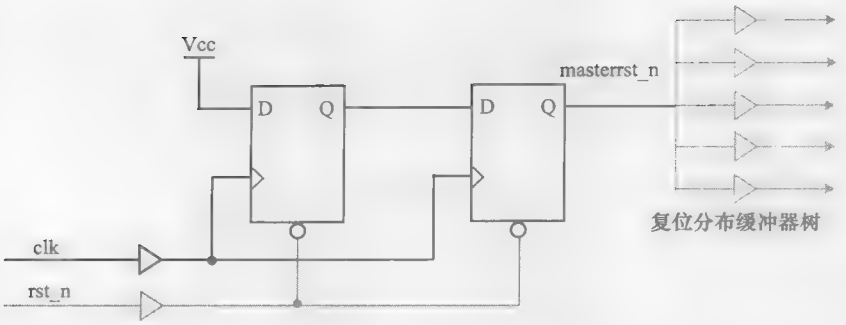


图 2.40 复位同步器模块图

通过撤销复位信号将复位移除，这时允许第一个主复位寄存器的 D 输入端信号在时钟控制下穿过复位同步器。在复位撤销后，典型情况下需要两个时钟上升沿来同步移除主复位信号。

将复位信号与时钟脉冲同步需要两个触发器，这里第二个触发器用于移除由于异步撤销的复位信号与时钟上升沿过于接近所导致的亚稳态。

还要注意的一点是在移除复位后，第二个触发器不存在亚稳态问题。复位同步器的第一个触发器有潜在亚稳态风险的原因是其输入固定为高电平，输出异步复位为 0 并且复位可能在触发器规定的恢复时间内移除（复位可能与同一个触发器的时钟输入上升沿过于接近）。这就是需要第二个触发器的原因。

复位同步器中的第二个触发器不会出现恢复时间亚稳态问题，因为在移除复位时该触发器的输入和输出都为低电平。由于这个触发器的输入和输出没有差别，因此其输出也就没有机会在不同逻辑值之间振荡。下面的等式计算了总的复位分布时间：

$$T_{\text{rst\_dis}} = t_{\text{clk-q}} + t_{\text{pd}} + t_{\text{rec}}$$

式中 $t_{\text{clk-q}}$ = 复位同步器中第二个触发器的时钟端到 Q 端的传播延迟

t_{pd} = 通过复位分布网络树的总延迟

t_{rec} = 目标触发器的恢复时间

2.6.6 过滤复位毛刺

异步复位对毛刺很敏感，这就意味着任何满足触发器最小复位脉冲宽度的输入都能引起触发器复位。如果复位线受到毛刺的影响，这就真的成

为问题了。在设计中，可能没有足够高频的采样时钟来检测复位上的小毛刺；本节将会介绍过滤掉毛刺的方法[30]。该方法需要一个数字延时来过滤毛刺。复位输入引脚也必须是施密特触发器引脚才有助于毛刺过滤。图2.41显示了复位毛刺滤波器的电路和时序图。

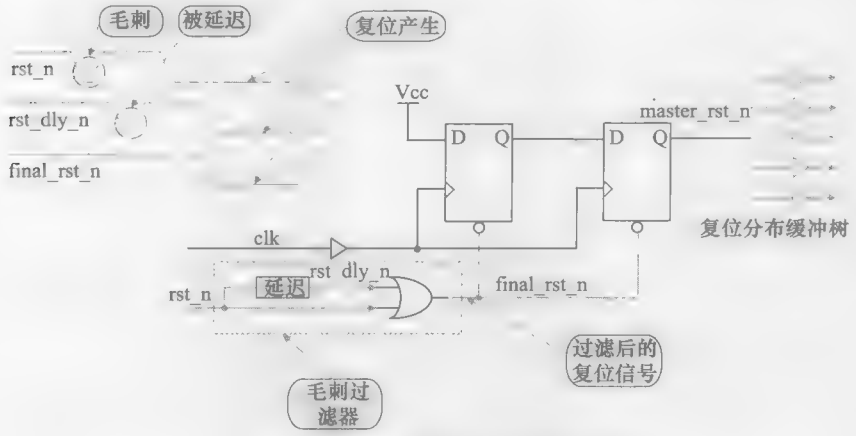


图2.41 复位毛刺过滤

为了加入延时，一些生产商提供了用于延迟且能够手动实例化的宏单元。如果没有这样的宏单元，设计人员就需要在优化后的已综合设计中手动加入延时。第二种方法需要创建一个包含较慢缓冲器的模块，再多次实例化该模块以达到所期望的延迟。基于这种思想，可以产生许多变种的解决办法。

由于该方法使用了延迟链，因此一个缺点是所产生的延迟会随着温度、电压和工艺而变化。必须注意确保延迟在所有PVT环境下都能满足设计要求。

2.7 控制时钟偏移

整个芯片中时钟信号到达时间的差异称为时钟偏移。时序必须满足寄存器建立和保持时间的要求是基本的设计原则。数据传播延迟和时钟偏移都用于与之相关的计算。对于同一时钟边沿偏移较大的寄存器，如果顺序相邻，那么在向其提供时钟时，就会有违背时序的潜在风险，甚至使功能失效。这是ASIC设计失败最主要的原因。

图2.42是两个顺序相邻触发器时钟偏移的例子。

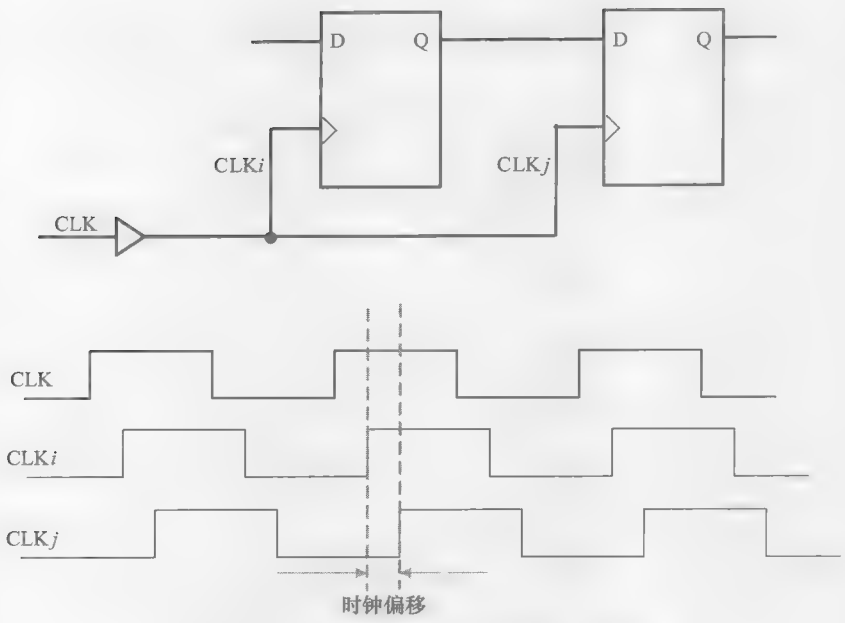


图 2.42 两个顺序相邻触发器的时钟偏移

给定两个顺序相邻的触发器 F_i 和 F_j 以及一个等电位时钟分布网络，这两个触发器之间的时钟偏移定义为：

$$Tskew_{i,j} = T_{c_i} - T_{c_j}$$

这里 T_{c_i} 和 T_{c_j} 分别是从时钟源到触发器 F_i 和 F_j 的延迟。

2.7.1 短路径问题

时钟偏移中的短路径问题与触发器的保持时间违背很相似。在两个相邻触发器之间的数据传播延迟比时钟偏移还短时，就会出现该问题。

图 2.43 中的电路图和时序图可以说明短路径问题。

由于同一个时钟沿到达第二个触发器比新数据要慢，因此第二个触发器在与第一个触发器同样的边沿处，切换为与第一个触发器同样的值。这会使 U2 在与 U1 同一个边沿处移位同样的数据，最终导致功能错误。

2.7.2 时钟偏移和短路径分析

之前提到，在两个顺序上相邻的触发器之间的数据路径传播延迟比两

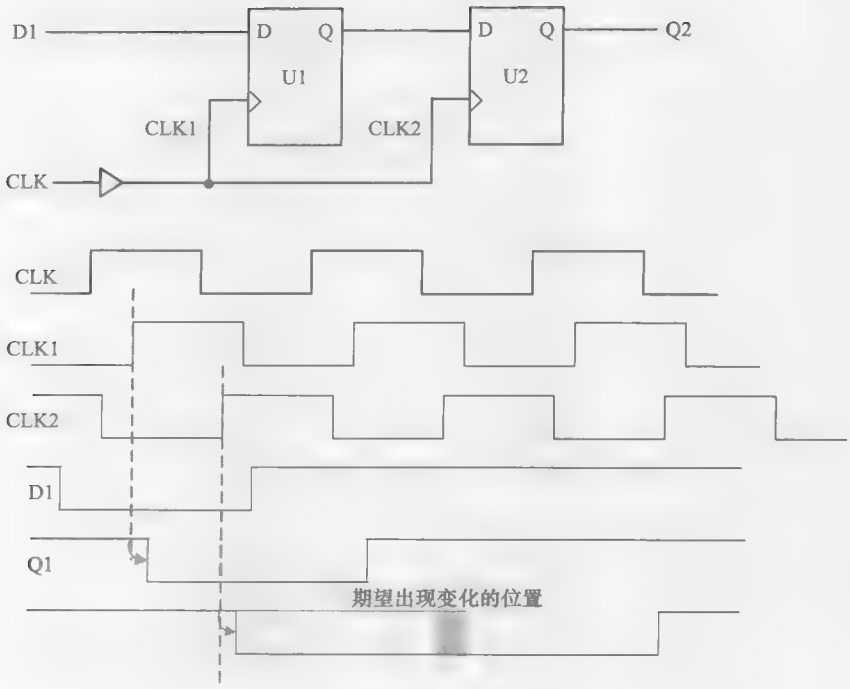


图 2.43 含短路径问题的电路

者之间的时钟偏移小时，会出现时钟偏移和短路径问题。

图 2.44 中的延迟如下所示。

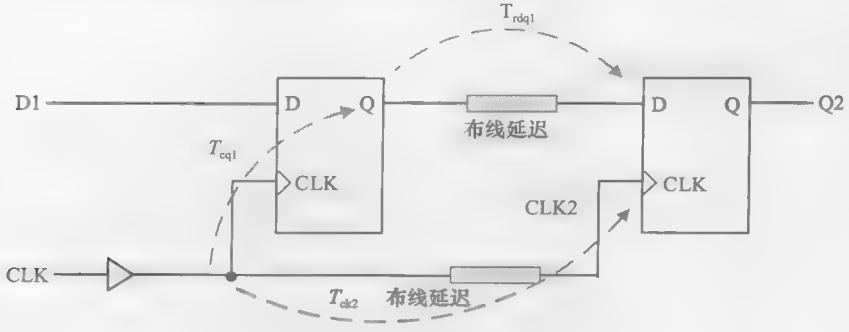


图 2.44 简单电路中的通用延迟模块

T_{cq1}：第一个触发器的时钟输出延迟。

T_{rdq1}：从第一个触发器的输出到第二个触发器输入的传播延迟。

T_{ck2}：第二个触发器的时钟到达时间与第一个触发器的时钟到达时间

之差。

在以下条件下，会出现明显的短路径问题

$$T_{ck2} > T_{cq1} + T_{rdq1} - T_{HOLD2}$$

这里 T_{HOLD2} 是第二个触发器的保持时间。

该区域如图 2.45 所示。

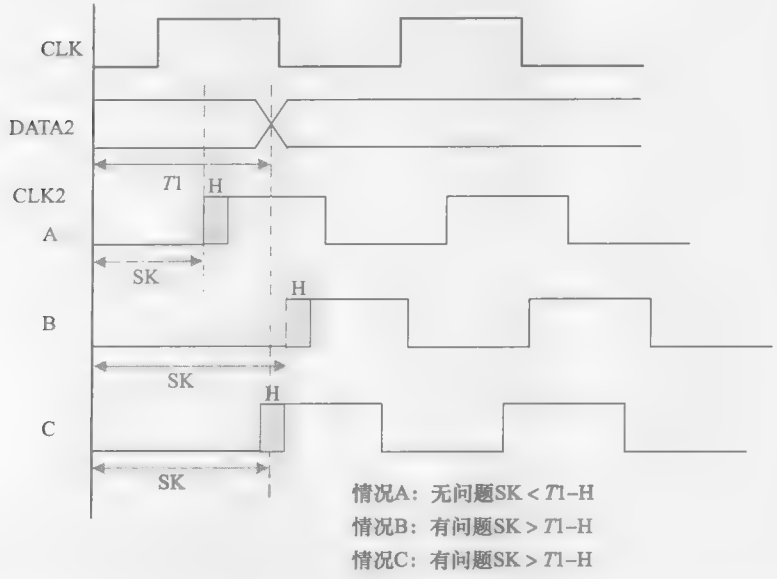


图 2.45 短路径问题图解

因此，为了识别带有该问题的路径，用户必须提取出时钟偏移（如 T_{ck2}）和短路径延迟（如 $T_{cq1} + T_{rdq1} - T_{HOLD2}$）。

2.7.3 使时钟偏移最小化

将时钟偏移减至最小值是解决短路径问题的最好方式。将设计中的时钟偏移保持在触发器的最小延迟之下能提高设计对所有短路径问题的健壮性。

下面几节介绍一些众所周知的设计技术以使设计对时钟偏移更健壮。

2.7.3.1 在数据路径上加入延迟

如图 2.44 所示，通过在数据路径上增加延迟（T_{rdq1}）而最终使整个数据路径的延迟大于时钟偏移，可以消除短路径问题。

在数据路径中插入的延迟必须足够大以保证数据路径延迟一定能大于时钟偏移。

2.7.3.2 时钟反转

时钟反转是避免出现短路径和时钟偏移问题的另一种方法。在这种技术中，将时钟相对于数据反转使用，这样就自动消除了时钟偏移。

在发送寄存器接收到时钟沿前，时钟会先驱动接收触发器读入发送（源）值。图2.46是实现时钟反转方式的一个小例子。

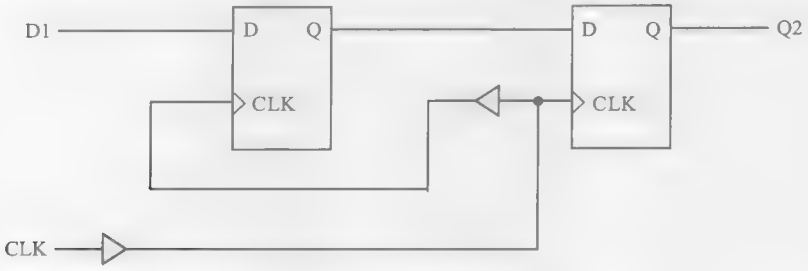


图2.46 时钟反转方法学

可以看出，在插入足够多的延迟后，接收触发器会比源触发器先接收到有效时钟沿。这是一种以牺牲建立时间为代价提高保持时间的方法。

时钟逆转方法对于约翰逊计数器和线性反馈移位寄存器（LFSR）这样的环路结构并不是很有效，因为对这种类型很难明确定义下一级触发器。图2.47是一个使用时钟逆转互联的环路结构，可以看出，在U1和U3触发器之间存在短路径问题。

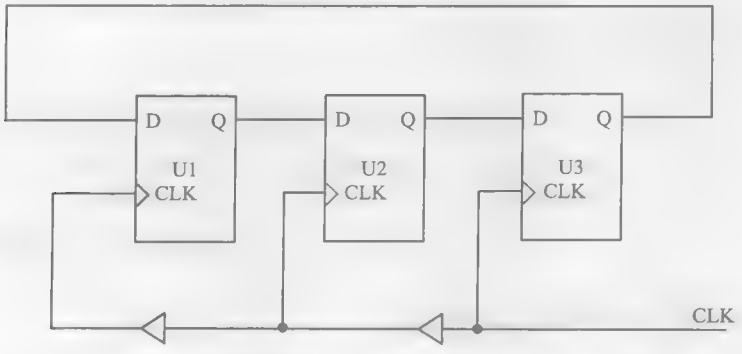


图2.47 环路结构中的时钟反转

2.7.3.3 交替相位时钟

避免时钟偏移问题的已知方法之一是使用交替相位时钟。下面几节介绍几种交替相位时钟的设计技术。

交替使用时钟沿

在这种方法中，顺序上相邻的触发器使用相反的时钟沿触发，如图2.48所示。

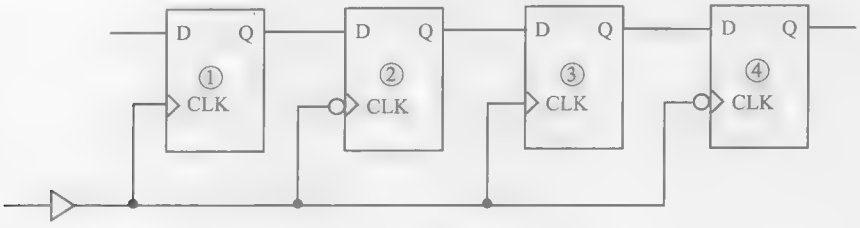


图2.48 交替边沿时钟

这种方法为时钟偏移提供了约半个时钟周期的短路径时钟偏移余量。

交替使用时钟相位

图2.49中的相邻触发器组分别由同一时钟的两个不同相位驱动。在这种情况下，任意两个相邻的触发器之间都有与两个相位的相位差大致相同的安全余量。

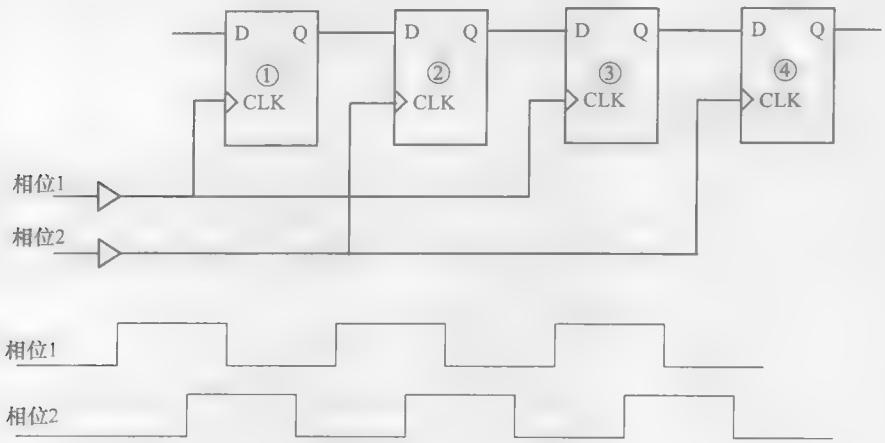


图2.49 交替相位时钟

必须注意的一点是，交替使用时钟相位要求对原始时钟信号使用完全不同的时钟约束。例如，在交替使用时钟边沿时，由于相邻触发器由同一个时钟周期相反的边沿驱动，因此时钟频率的新约束值应该时初始约束频率的一半。

行波时钟结构

在行波结构中，每个触发器的输出驱动下一个触发器的时钟端，这与行波计数器的实现方式类似。只有在源触发器翻转时，下一级触发器才会被时钟驱动，如图 2.50 所示。

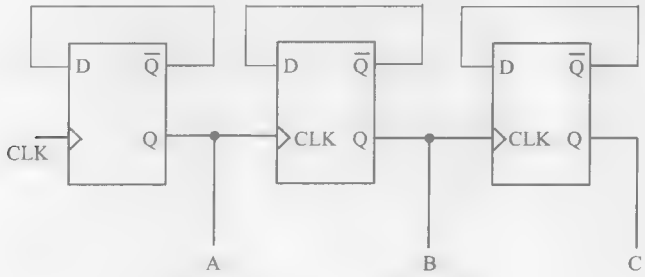


图 2.50 三位行波逐减计数器

在图 2.50 中，每个计数触发器的输出驱动下一个触发器的时钟端，而不是数据输入端。由于触发器不在同一个时钟翻转，这就消除了时钟偏移。第一个触发器由 CLK 信号的正沿驱动，第二个和第三个触发器由前一个触发器输出信号的正沿驱动。

根据设计复杂度和所使用的方法学，可以使用上面所提到的不同技术来最小化时钟偏移和避免短路径问题。

2.7.3.4 平衡线路长度

前面几节提到的技术都应在项目前期就计划使用。当然作为上述方案的替代，设计者也可以使用平衡线路长度的方式保证低时钟驱动器偏移。除了保持所有时钟线长度相等，也应保证各终端负载相同。这样可以保证线路长度恰当地平衡。

下面是应当遵守的一些指南。

1）紧密关注驱动器上的输入－输出延迟规范。

2）在时钟各级层次上使用同样的驱动器。

3）平衡各级标称的线路延迟。

4）在各条线路上使用同样的终端策略。

5）平衡各条线路的负载，这可能意味着需要在一条支路上添加多余电容以与其他分支平衡。

参考文献

1. Mohit Arora, Prashant Bhargava, Amit Srivastava, *Optimization and Design Tips for FPGA/ASIC(How to make the best designs)*, DCM Technologies, SNUG India, 2002
2. Application Note, ASIC design guidelines, Atmel Corporation, 1999
3. Cummings CE, Sunburst Design, Inc.; Mills D, LCDM Engineering (2002) Synchronous resets? Asynchronous resets? I am so confused! How will I ever know which to use? SNUG, San Jose
4. Application Note, Clock skew and short paths timing, Actel Corporation, 2004

第3章

处理多个时钟

3.1 介绍

只涉及单个时钟的设计是易于实现的。但是在实践中，很少有设计只在一个时钟下运作。本章将会涉及使用多个时钟的设计，并介绍在设计过程中所遇到的问题和处理方法，最后得到可以工作在多个时钟下的健壮设计。

单时钟设计（更确切地说，也就是同步设计）如图3.1所示。在单时钟域中，有单个时钟贯穿整个设计。同多时钟设计相比，这样的设计更易于实现，并且更少产生与亚稳态、建立和保时间违背方面的问题。

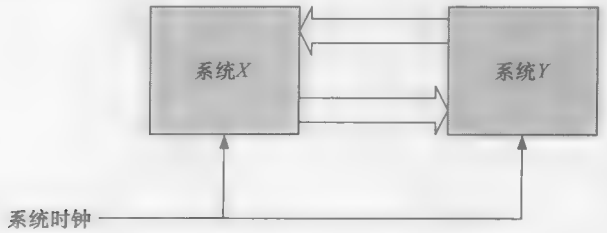


图3.1 单时钟域

3.2 多时钟域

对于工程师来说，开发含多个时钟（见图3.2）的设计是一种挑战。

这样的设计中可能有以下任何一个，或者全部类型的时钟关系：

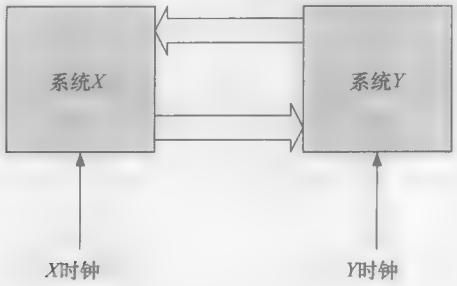


图 3.2 多时钟域

- 时钟的频率不同
- 时钟频率相同，但相位不同

以上两种关系如图 3.3 所示。

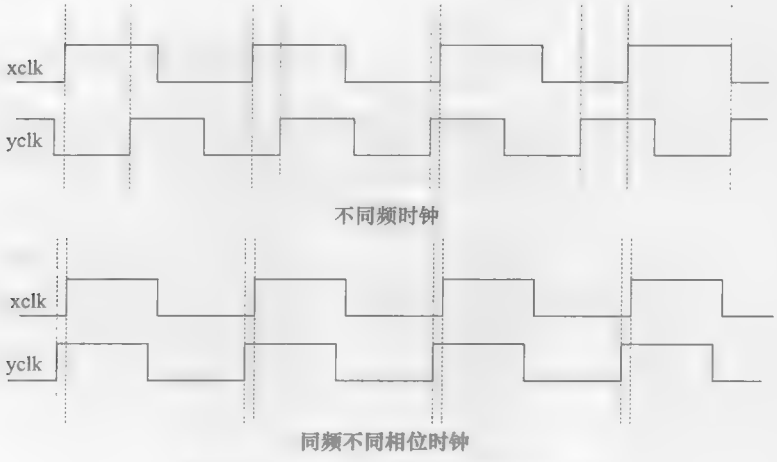


图 3.3 多时钟间关系（在多时钟域中）

3.3 多时钟域设计的难题

多时钟设计将面临下列问题：

- 建立时间和保持时间的违背
- 亚稳态

3.3.1 违背建立时间和保持时间

建立时间：在时钟脉冲到来前，输入数据需要保持稳定的时间。
保持时间：在时钟脉冲到达后，输入数据仍需保持稳定的时间。
图 3.4 解释了相对于时钟上升沿的建立时间和保持时间。

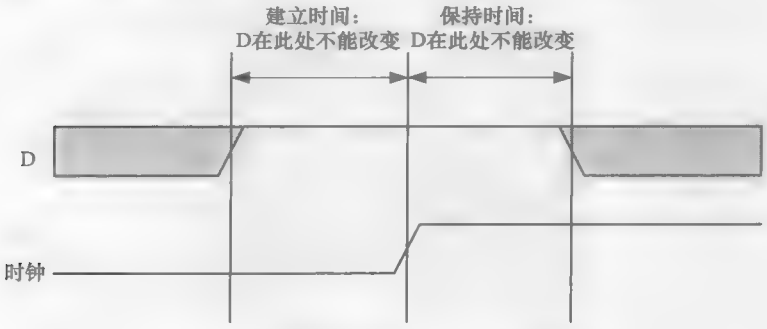


图 3.4 建立和保持时间（相对于时钟边沿）

建立时间要求数据必须在时钟上升沿到来前保持稳定，保持时间要求在时钟上升沿到来后数据必须仍然保持稳定。对于单时钟域，这样的要求很容易满足。但是，在多时钟域情况下，很容易出现一个时钟域的输出在另一个时钟域中的时钟上升沿到来时发生改变的现象。这将会引起第二个时钟域中触发器的输出处于亚稳态，由此导致一系列错误的结果。

图 3.2 中所示的是一个双时钟系统，图 3.5 中所示的是这样一个系统中信号跨时钟域传输的时序图。如图 3.5 所示，xclk_output1（属于 xclk 时

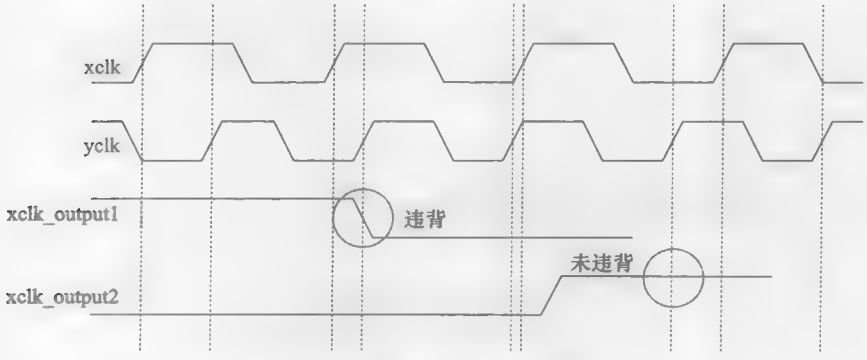


图 3.5 建立和保持时间违背

钟域）在 yclk 的上升沿附近发生了变化。即 xclk_output1 信号在变化期间被 yclk 时钟域采样。这会导致相对于 yclk 的建立时间和保持时间违背现象的发生。因此，在 yclk 时钟域中，依赖于 xclk_output1 的信号将进入亚稳态并产生错误的结果。然而，xclk_output2（属于 xclk 时钟域）在 yclk 的上升沿处是稳定的。所以不会出现建立时间和保持时间违背的问题。因此，对于 yclk 时钟域中的信号，如果依赖 xclk_output2 信号，将会产生正确的输出。

3.3.2 亚稳态

由多个时钟域引起的亚稳态问题已经在第 1 章中做了详细的介绍。

后序章节将介绍一些设计技术和具体的解决方法，来确保得到一个稳定可靠的多时钟域设计。

3.4 多时钟设计的处理技术

在进行一个含多个时钟的设计时，在仿真和综合过程中遵循一定的准则将会带来很大的好处。一些通用的准则如下：

- 时钟命名规则
- 分模块设计

3.4.1 时钟命名法

为了保证综合脚本可以更轻松地处理所有的时钟信号，有必要对整个设计使用一个确定的命名步骤。例如，系统时钟可以命名为 sys_clk，发送时钟可以命名为 tx_clk，接收时钟可以命名为 rx_clk 等。这样就可以在脚本中使用通配符来对所有时钟进行操作。同理，属于同一个时钟域的信号，也应该在命名时使用同样的前缀。例如，由系统时钟驱动的信号，可以用类似于 sys_rom_addr、sys_rom_data 这样的方式作为起始。

使用这种命名法，团队中的每个工程师便可以分辨出每个特定的信号所属的时钟域，并决定是应该直接使用该信号，还是需要先将其通过同步器后再使用。

这样的命名过程能极大地减少对信号的混淆，并在各模块之间提供简单的接口，因此提高了团队的工作效率。

3.4.2 分块化设计

这是设计含多时钟的模块时常用的另一种有效技术。如下所述：

1）每个模块只应当在单个时钟下工作。

2）在信号跨时钟域传输时，使用同步器模块（该模块的作用是将信号从一个时钟域转递到另一个时钟域），以使所有信号在进入某个时钟域内的模块时，与该模块的时钟保持同步。

3）同步器模块的规模应尽可能小。

将整个设计分割成模块的优点在于使得静态时序分析变得很简单，因为所有输出或输入某时钟域的信号都与使用该时钟的模块保持同步。所以，设计就成为完全同步的。另外，对于同步模块是不需要做静态时序分析的。但是，要保证满足保持时间的要求。

如图3.6所示，整个逻辑分成了三个时钟域，分别是Clock1时钟域、Clock2时钟域和Clock3时钟域。根据3.4.1节中所述的命名规则对各时钟域中信号进行命名。所有跨时钟域传输的信号都要经过一个额外的同步模块。该同步模块的作用是将信号从原来所在的时钟域传化到将使用该信号的时钟域。因此，如图3.6所示，“从1同步到2”模块将信号从Clock1时钟域转化到Clock2时钟域。

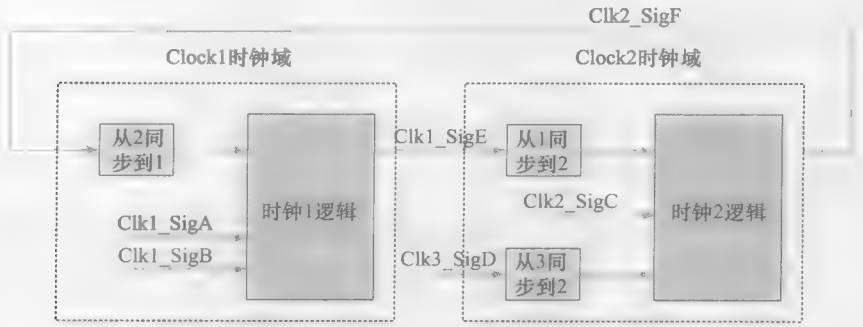


图3.6 设计划分

3.4.3 跨时钟域

跨时钟域信号的传输可以归为两类，分别是：

- 控制信号的传输

- 数据信号的传输

3.4.3.1 控制信号的传输（同步化）

在设计中，如果将一个异步信号直接送给若干个并行工作的触发器，就会大大增加亚稳态事件发生的概率，因为有可能有多个触发器进入亚稳态。为了避免形成这种情况下的亚稳态，我们常常使用同步触发器的输出信号来取代异步信号。

为了减少亚稳态的影响，设计者最常用的方式是使用多级同步器，即将两个或多个触发器串联起来组成的同步电路，如图 3.7 所示。

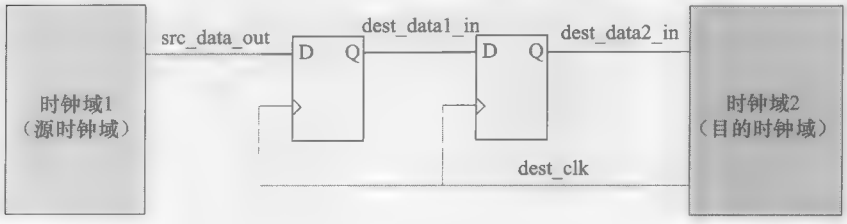


图 3.7 双级同步电路

如果同步器的第一级触发器产生亚稳态输出，那么这个亚稳态会在同步器的第二个触发器取样前进入稳态。这种方法无法保证第二级触发器的输出一定不会出现亚稳态，但是它确实降低出现亚稳态的可能性。同理，如果为同步器增加更多级触发器，就会进一步降低出现亚稳态的可能性。

这种方法的一个缺点，也可以看做同步器电路不可避免的开销，就是增加了电路的整体延时。

上述同步电路的时序如图 3.8 所示。

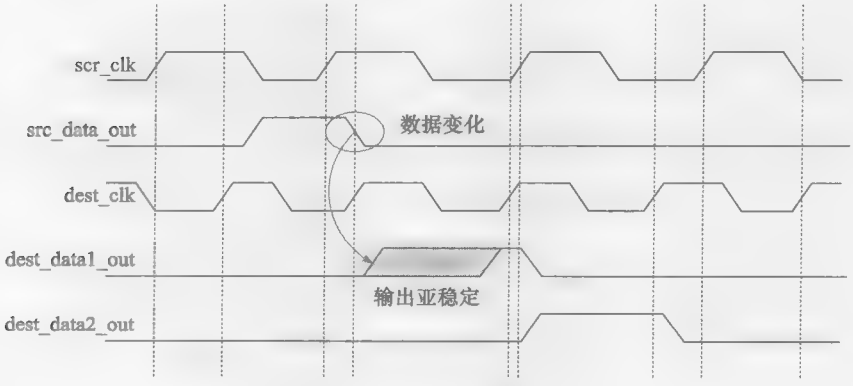


图 3.8 双级同步电路的时序

源时钟域使用 src_clk 作为时钟，其异步输出信号（src_data_out）输入第一个同步触发器中。dest_data1 _in 信号（即第一个同步触发器的输出）进入亚稳态，但是在第二个同步电路取样前会进入稳定状态。因此把 dest_data2_in 信号（第二个同步触发器的输出）同步化到目的时钟域的 dest_clk 时钟。

有些情况下，第一级同步器触发器的输出信号从亚稳态进入稳定状态所需的时间可能需要不止一个时钟周期，这就意味着第二级同步触发器的输出仍然亚稳定。这时为安全起见，应当使用三级同步器电路。

图 3.9 是一个三级同步器电路的例子。

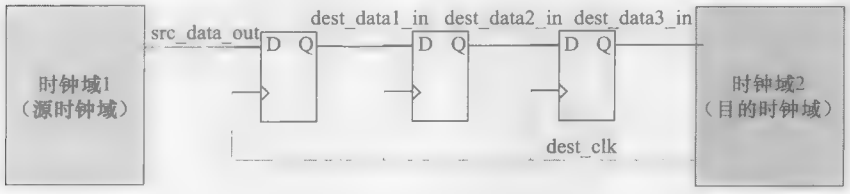


图 3.9 三级同步电路

三级同步器电路由三个串联的触发器组成。第二级触发器可能输出的亚稳态信号在第三级触发器采样前会进入稳定状态。

三级同步器电路的时序如图 3.10 所示。

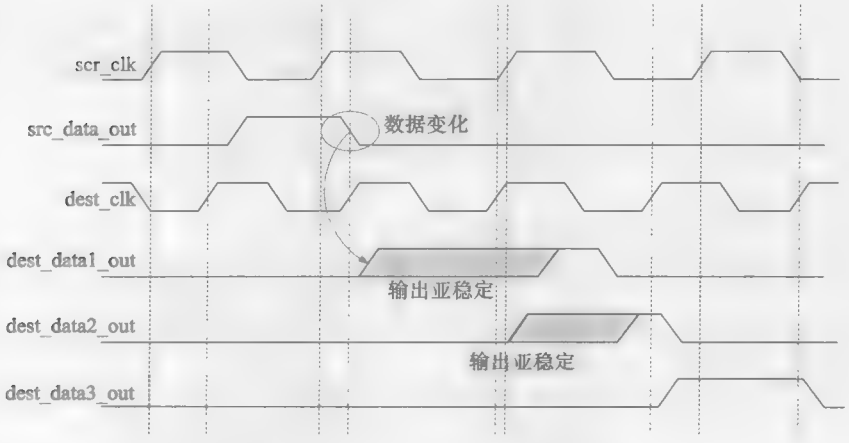


图 3.10 三级同步电路的时序

源模块的异步输出信号（src_data_out0）工作在 src_clk 时钟下，并送给第一个同步器触发器。信号 dest_data1 _in（第一个同步触发器的输出信

号）进入亚稳态，但可以在多个时钟周期的时间里恢复到稳定状态。在这段时间内，第二个触发器会对第一个触发器的输出进行采样，于是信号dest_data2_in（第二个同步器触发器的输出）也会进入亚稳态。如图所示，在第二个触发器取样前，该信号可以进入稳定状态。用这种方式可以将src_clk时钟域的异步输出信号同步到 dest_clk 时钟域中。

尽管如此，在大多数多时钟域设计中，使用两级同步电路就足以避免亚稳态的出现了。所以，只有在时钟频率非常高的设计中才要求使用三级同步器电路。

3.4.3.2 数据信号的传输

在多时钟域设计中，数据经常会从一个时钟域传输到另一个时钟域。以下是保证数据正常地在不同时钟域间传输的两种方法：

- 使用握手信号的方式
- 使用异步 FIFO

以上两种技术将在 3.6 节和 3.8 节中详细介绍。

3.5 跨时钟域

在第 4 章介绍各种在异步时钟域间传输数据的方法之前，让我们在本节中先来看看跨同步时钟域问题的各种类型。

如果多个时钟都起源于同一时钟，并且它们的相位和频率关系是已知的，那么这些时钟可以看成是跨同步时钟域的时钟。按照相位和频率的关系，可以将这些时钟分成以下类型：

- 同频零相位差时钟
- 同频恒定相位差时钟
- 非同频可变相位差时钟
  - 整数倍时钟
  - 有理数倍时钟

下一节假定两个时钟之间的相位和时钟抖动相同，并假定它们之间的路径已经按同样的时钟延迟和偏移参数进行了平衡。除此之外，还假设这两个时钟起始处的相位差为零，而且触发器的“时钟到 Q 端”的延时也为零。

3.5.1 同频零相位差时钟

在此种情形下，两个完全相同的时钟“clk1”和“clk2”具有同样的

频率与零相位差。“clk1”和“clk2”是完全相同的，并由同一个时钟源产生，在“clk1”和“clk2”之间的数据传输并不算是跨时域的。实际上，这种情况就是单时钟设计，在这里仅仅是出于全面考虑才提及它。

无论何时，从“clk1”到“clk2”的数据都有一个完整的有效时钟周期用于传输，以保证数据可以正常捕捉，如图3.11所示。

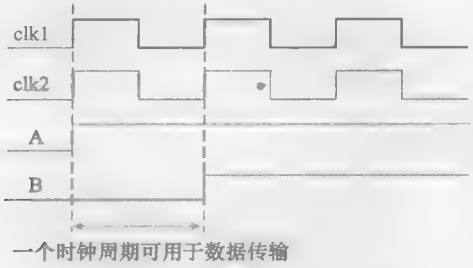


图3.11 频率和相位都相同的时钟

只要在源触发器和目的触发器之间的组合逻辑的延迟能满足电路建立和保持时间的要求，数据就能正确地传输。在这种情况下，对设计的唯一要求只是保证STA（静态时序分析）通过。如果这一条件满足，就不会出现亚稳态问题和数据丢失或不一致的问题。

3.5.2 同频恒定相位差时钟

这些时钟有相同的时钟周期，但是相位差恒定。典型例子是对某个时钟及其反相时钟的使用。另一个例子是某个时钟相对于其上级时钟发生了相位移动，例如，如果规定T为时钟周期，移动值为T/4。

如图3.11所示，时钟“clk1”和“clk2”的频率相同，但在图3.12中“clk1”相对于“clk2”相位前移了3T/4个单位时间。

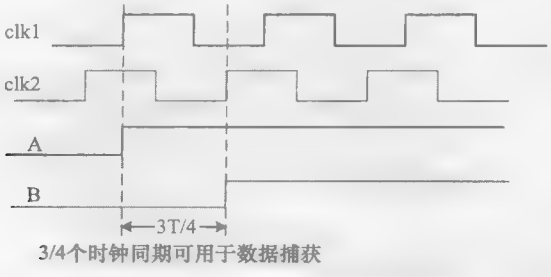


图3.12 频率相同但相位移动的时钟

每当数据从“clk1”传输到“clk2”时，由于更小的建立时间/保持时间裕量，对组合逻辑的延时约束都会变得更紧。如果组合逻辑的延时能保证满足采样沿处建立时间和保持时间的要求，数据就可以正确地传输，并且不会有亚稳态产生。在这种情况下是不需要同步器的。只需要使设计的STA通过就可以了。

一般会在STA中创建这种情况以保证满足时序要求。如果组合逻辑有更多延时，通过在发射边沿和捕获边沿加入偏移（例如，使时序有相同频率和不同相位），会有助于满足时序的要求。

3.5.3 非同频、可变相位差时钟

这类时钟具有不同的频率，而且相位差也是可变的。可以将这类时钟分为两个子类，第一子类中各时钟周期之间是整数倍的关系，第二子类中各时钟周期是非整数倍（有理数倍）的关系。不论是哪一子类，各时钟有效边沿的相位差都是可变的。下面将会详细介绍这两种情况。

3.5.3.1 整数倍频率的时钟

在这种情况下，一个时钟的频率是另一个时钟的整数倍，并且它们有效边沿的相位差是可变的。这两个时钟的有效边沿之间可能的最小相位差始终等于其中较快的那个时钟的时间周期。

在图3.13中，时钟“clk1”的频率是时钟“clk2”的三倍。假定T是时钟“clk1”的时钟周期，时钟“clk2”可以用来捕获数据的时间可能是T、2T或3T，这取决于数据在时钟“clk1”的哪个边沿发出。因此，任意路径的最差延迟都应在时钟边沿相位差为T时满足建立时间的要求。最差保持时间检查应当在时钟边沿相位差为零时进行。

在以上所有情况下，都有较快时钟的一个完整的周期用来传输数据，以使其能正确地捕捉，所以它通常都可以保证满足建立时间和保持时间的要求。因此，就不会存在亚稳态或数据不一致的问题，也就不必使用同步器了。

既然在这里数据由较快时钟发出，并由较慢时钟捕获，那么为了避免数据丢失，源数据应该保持至少一个目的时钟周期的稳定状态。可以使用一些控制电路来满足这一要求，例如，使用一个简单的有限状态机（FSM）。参考图3.13，如果使源数据每三个源时钟改变一次，就不会出现丢数据的问题了。

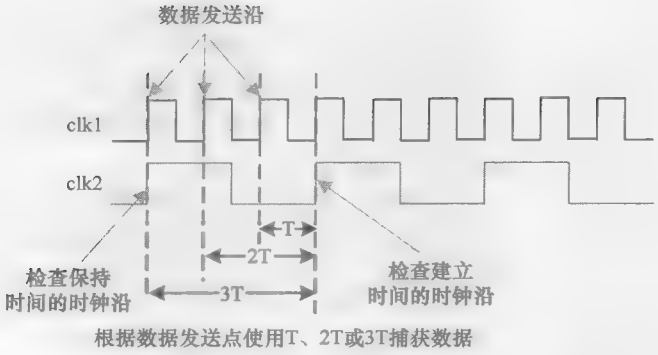


图 3.13 含内部倍频的时钟

3.5.3.2 非整数倍频率的时钟

这种情况指一个时钟的频率是另一个时钟的非整数倍，而且它们有效边沿的相位差是可变的。

与前面提到的各时钟之间保持整数倍关系不同，因此两时钟之间的最小相位差足以使亚稳态产生。至于亚稳态是否真的发生，取决于实际的频率倍数和设计工艺。下面介绍三种不同的情况。

在第一种情况下，在源时钟有效沿和目的时钟有效沿之间有足够大的相位差，所以不会有亚稳态产生。

在第二种情况下，源时钟和目的时钟有效沿非常接近，导致产生亚稳态问题。然而，在这里时钟频率倍数关系需要满足以下条件，即一旦有时钟边沿接近出现，下一个时钟周期就会留出足够大的时间冗余，使得数据的捕获不会出现违背建立时间或保持时间的要求。

在第三种情况下，两个时钟的时钟沿在许多连续的周期中都非常接近。这与异步时钟的行为很相似，但因为这两个时钟的源头是相同的，所以它们之间的相位差是可以计算出来的。

注意，在下面的所有例子中，会使用某些延迟值，假设时钟边沿之间的相位冗余在小于或等于 1.5ns 时会产生亚稳态。这里的 1.5ns 并只是一个比喻，在实际设计中与所用技术、触发器特征等许多因素相关。

例子 1

在这种情况中两个时钟的有效沿永远不会过于接近，从而可以保留足够的冗余来满足电路对建立时间和保持时间的要求。

假设两个时钟“clk1”和“clk2”分别是对同一个时钟“clk”的 3 分频与 2 分频。那么“clk1”比“clk2”慢 1.5 倍。如图 3.14 所示，“clk1”

的周期是 15ns，“clk2”的周期为 10ns。这两个时钟间的最小相位差是 2.5ns，这对于满足建立时间和保持时间已经足够了。

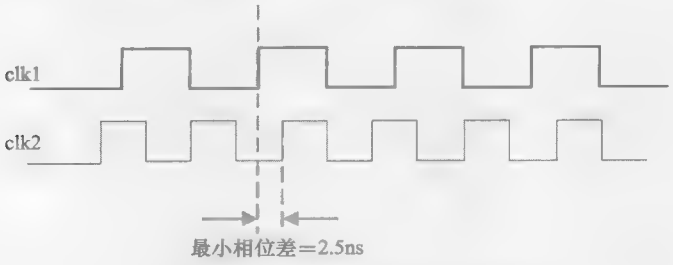


图 3.14 时钟沿相距较远所以避免了亚稳态

然而，由于该相位差很小，应该避免在跨越两个时钟的位置使用任何组合逻辑。对于增加的任何组合逻辑，必须满足建立和保持时间的要求以避免亚稳态，因此必须使用同步器。

进一步说，在数据从慢时钟域传递到快时钟域时，必须增加逻辑以保证数据在快时钟域中只取样一次，这时不会有数据丢失的现象。然而，在从快时钟域向慢时钟域传递数据时，就可能出现数据丢失。为了解决这个问题，必须将源数据保持至少一个目标时钟周期，以保证在两个连续变换的源数据之间至少有一个目标时钟到达。

例子 2

在这种情况下，两个时钟的有效沿可能间隔性地非常接近。换句话说，两个时钟沿会出现挨着的情况，然后在再次出现挨着的情况之前，接来的几个周期两个时钟沿会保留足够的裕量（能正确捕捉到数据）。这里“挨着”的意思是接近到了足以产生亚稳态的程度。

在图 3.15 中，时钟“clk1”和“clk2”的周期分别为 10ns 与 7ns。注意，它们之间的最小相位差是 0.5ns，这已经非常小了。所以会出现亚稳态，并且必须使用同步器。

由于亚稳态的出现，当两个时钟沿很接近时数据不能被目标域正确地捕获。然而，在这种情况下，注意，一旦出现时钟沿非常接近的情况，下一个周期的裕量就会很大，以便数据就可以被目标时钟正确捕获。图 3.15 中的信号 B2 就表示这种情况。所期望的输出是 B1，实际的波形是 B2。注意，这里数据不会丢失，但是可能有数据不连贯的问题。

对于从快时钟域到慢时钟域的传输，可能出现数据丢失，为了阻止这种现象，源数据应该保持至少一个目标时钟周期不变。可以通过使用一个简单的 FSM 实现这一目的。

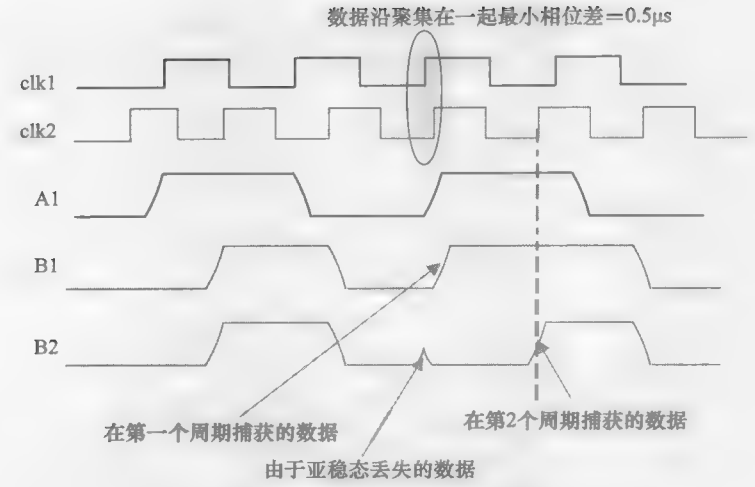


图 3.15 间歇性聚集的时钟沿

例子 3

在这种情况下时钟间的相位差异很小，并能连续存在几个周期。除了变化的相位差异和周期性的重复现象，其余都与异步时钟很相似。

在图 3.16 中，时钟“clk1”和“clk2”的周期分别为 10ns 与 9ns。可以看出两个时钟的有效时钟沿很接近，并持续 4 个连续周期。在前两个周期中可能会违背建立时间（源时钟在目的时钟之前），而在后两个时钟周期中可能会违背保持时间（目的时钟在源时钟之前）。

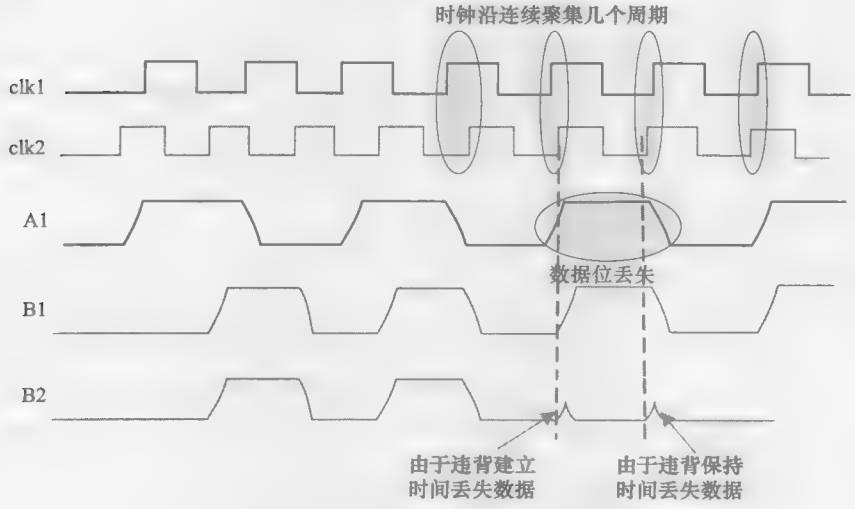


图 3.16 连续聚集几个周期的时钟沿

在这种情况下，将会出现亚稳态问题，因此需要进行同步。除了亚稳态的问题，数据从慢时钟域传递到快时钟域时也可能失丢。可以从图 3.16 中可以看到，B1 是不含亚稳态的正确输出。但实际的输出可能是 B2。这里数值 1 丢失了，因为第一个周期“1”由于违背建立时间未能捕捉到，而在第二个周期中由于违背保持时间误捕捉到了“0”。

为了不丢失数据，数据需要保持稳定至少两个目的时钟周期。这既适用于从快到慢的传输，也适用于从慢到快的传输。可以通过使用简单的 FSM 对源数据产生进行控制完成这一任务。但数据不连贯的问题仍然存在。

这时，已标准化的技术（如握手和 FIFO）对于传输数据更为有效，因为它们也解决了数据不连贯的问题。

3.6 握手信号方法

使用握手信号是最古老的在不同域之间转输数据的方式。

图 3.17 是由两个时钟域分割成的两个单独的系统。

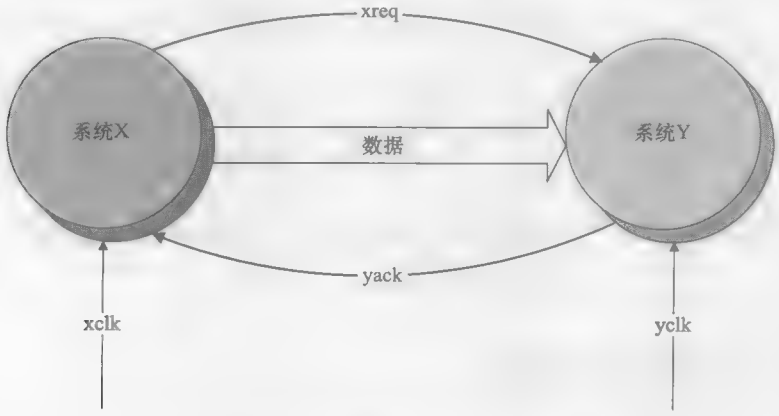


图 3.17 将双时钟域划分为两个独立的系统

使用握手信号“xack”和“yreq”，“系统 X”将数据发送给“系统 Y”。

下面是使用握手信号传输数据的例子。

1）发送器“系统 X”将数据放到数据总线上并发出“xreq”（请求）信号，表示有效数据已经发到接收器“系统 Y”的数据总线上。

2）把“xreq”信号同步到接收器的时钟域“yclk”上。

3）接收器在识别“xreq”同步的信号“yreq2”后，锁存数据总线上的信号。

4）接收器发出确认信号“yack”，表示其已经接受了数据。

5）接收器发出的“yack”信号同步到发送时钟“xclk”上。

6）发送器在识别同步的“xack2”信号后，将下一个数据放到数据总线上。

握手信号序列的时序如图3.18所示。

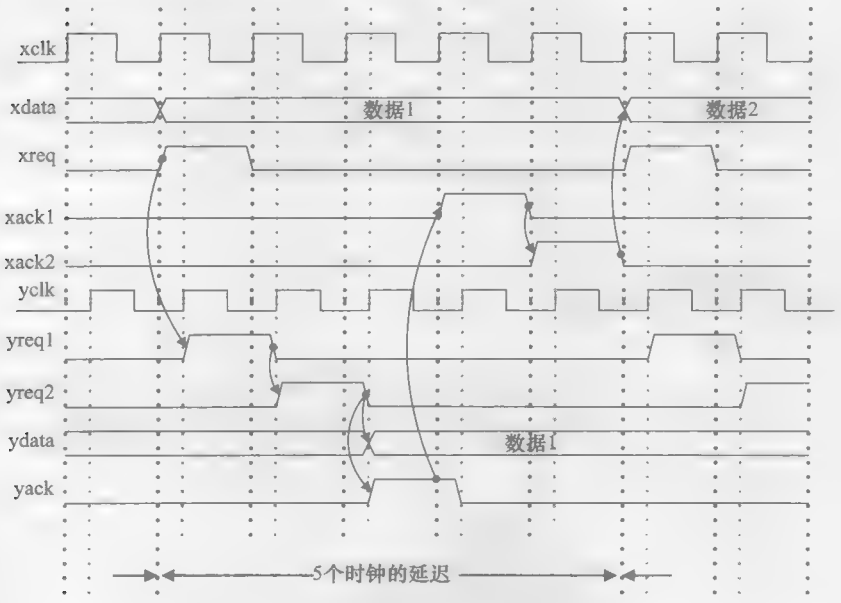


图3.18 使用握手机制传输数据的时序图

在图3.18中可以看出，安全地将一个数据从发送器传输到接收器需要5个时钟周期。

3.6.1 握手信号的要求

数据应该在发送时钟域内稳定至少两个时钟上升沿。

请求信号“xreq”的宽度应该超过两个上升沿时钟，否则从高速时钟域向低速时钟域传递可能无法捕捉到该信号。

3.6.2 握手信号的缺点

跨时钟域传输单个数据的延迟比使用FIFO（在后面章节中会介绍）

传输相同的数据要大得多。

3.7 使用同步 FIFO 传输数据

在设计系统时，会包含工作在不同时钟频率下的元件，例如，处理器、外设等。有时它们可能有自己的时钟振荡器。数据在这些元件之间传输时先进先出（FIFO）阵列起到了重要作用。FIFO 是用于对在通信总线上传输的数据进行排列的简单存储结构。

因此，FIFO 常用来传输跨不同时钟域的数据。

本节介绍简单的同步 FIFO 架构，其读和写使用同样的时钟。后面几节会详细介绍读、写分别使用不同时钟频率的异步 FIFO。

3.7.1 同步 FIFO 架构

图 3.19 展示了一个同步 FIFO 的通用架构。DPRAM（双端口 RAM）用作 FIFO 的存储器以使读、写可以独立进行[45]。

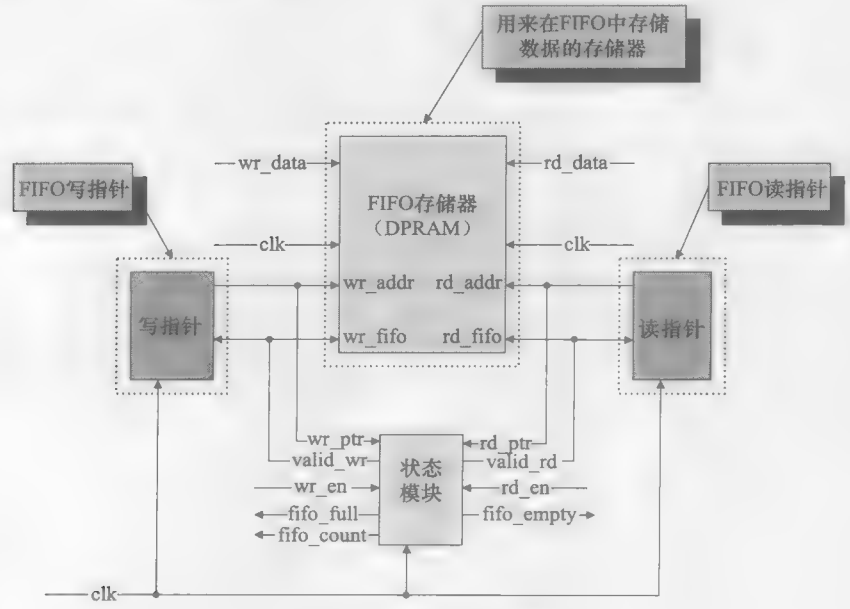


图 3.19 同步 FIFO 架构

通过读、写指针产生各自的读、写地址，送到读、写端口。写指针指向下一个要写入的地址，读指针指向下一个要读取的地址。有效写使能使写指针递增，而有效的读使能使读指针递增。

图3.19中的“状态模块”“产生”fifo_empty”和“fifo_full”信号。如果“fifo_full”有效，说明FIFO内的空间已满不能再写入数据。如果“fifo_empty”有效说明FIFO内没有可供读取的下一个有效数据。通过对两个指针进行相同的逻辑，该模块也指示出任意时刻FIFO中空或满区域的个数。

图3.13中所示的双端口存储器（DPRAM）可以同步读取或异步读取。对同步读操作，应该在数据在FIFO输出端有效前提供明确的读信号。对于异步读操作，DPRAM没有寄存的数据输出；有效数据在写入后即可用（先读数据然后递增指针值）。

3.7.2 同步FIFO的工作方式

复位后读写指针都归0。此时“fifo_empty”信号置为有效而“fifo_full”保持低电平。因为FIFO为空，所以阻止对FIFO的读操作，只能进行写操作。后序的写操作会增加写指针的值，并将“fifo_empty”信号置为无效。在没有空间可写数据时，写指针等于RAM_SIZE-1。此时进行一个写操作会使写指针回滚到0，并将“fifo_full”信号置为高电平。

总之在读、写指针相等时，FIFO要么空要么满，所以需要对这两种情况进行区分。

3.7.2.1 FIFO空和满的产生

图3.20显示深度为4的FIFO出现满的情况。

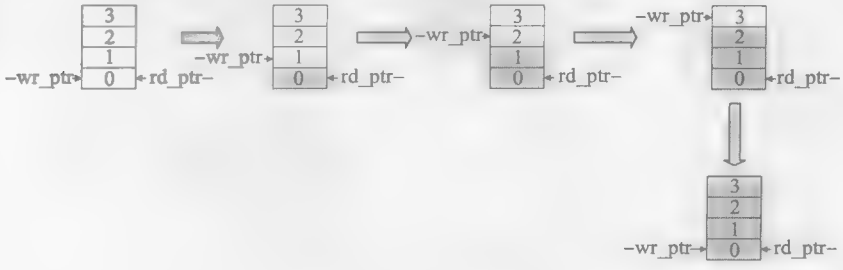


图3.20 FIFO满条件

图3.20中的转换发生在随后的时钟中。如图3.20所示，在写操作使

两个指针在下个时钟保持相等时，FIFO 满。这使得在以下情况会发出“fifo_full”信号。

$$\text{fifo\_full} = (\text{read\_pointer} = = (\text{write\_pointer} + 1)) \text{ AND" write"} \quad (3.1)$$

下面提供了产生“fifo_full”逻辑的 Verilog 代码。

```
always @ (posedge clk or nededge reset_n)
  begin: fifo_full_gen
   if (~reset_n)
      fifo_full < = 1'b0;
   else if (wr_fifo && rd_fifo)
              ;//do nothing
   else if (rd_fifo)
        fifo_full <= 1'b0;
   else if (wr_fifo && (rd_ptr = wr_ptr + 1'bl))
        fifo_full <= 1'bl;
  end
```

类似地，当读操作使两个指针在下一个时钟相等时，FIFO 变空。以下情况会产生“fifo_empty”信号。

$$\text{fifo\_empty} = (\text{write\_pointer} = = (\text{read\_pointer} + 1)) \text{ AND" read"} \quad (3.2)$$

下面提供了产生“fifo_empty”逻辑的 Verilog 代码。

```
always @ (posedge clk or negedge reset_n)
  begin: full_gen
   if (~reset_n)
    fifo full < = 1'bl;
   else if ( wr_fifo && rd_fifo)
         ; //do nothing
   else if (wr_fifo)
    fifo_empty <= 1'b0;
   else if (rd_fifo && (rw_ptr = rd_ptr + 1'bl ))
    fifo_empty <= 1'bl;
  end
```

3.7.2.2 另一种方法

另一种产生“fifo_full”和“fifo_empty”状态的方法是使用计数器来持续指示 FIFO 中空或满位置的个数。

计数器的宽度要与 FIFO 的深度相等以使其能记录 FIFO 中数据的最大个数。计数器在复位时初始化为 0。随后的任何写操作会将其递增 1，任何读操作会使其递减 1。

现在，在计数器值为 0 时，很容易就判断 FIFO 处于空状态，而当计数器的值等于 FIFO 的大小时，就能判断出 FIFO 处于满状态。

本节提到的这种方法虽然在原理上简单，但是与 3.7.2.1 节中所提到的方法相比效率要低一些。因为这种方法要求增加额外的硬件（比较器）

来产生 FIFO 空和满的条件。随着 FIFO 深度的增加，比较器的宽度也会增加，因此产生 FIFO 空、满信号需要更高级的序列比较器。这最终会降低 FIFO 操作的最高频率。

3.8 异步 FIFO（或双时钟 FIFO）

异步 FIFO 用来在两个异步时钟域间传输数据。

图 3.21 中是两个系统，分别为“System X”和“System Y”，从“System X”向“System Y”传输数据，两个系统工作在不同的时钟域中。

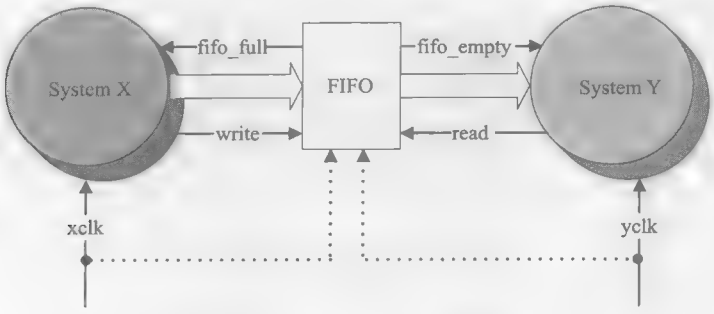


图 3.21 用异步 FIFO 传输数据

“System X”使用“xclk”将数据写入 FIFO，并由“System Y”使用“yclk”读出。

“fifo_full”和“fifo_empty”信号负责对上溢和下溢情况的监控。

“fifo_full”信号指示上溢情况。在“fifo_full”置起时数据不应写入 FIFO，否则会将 FIFO 内的数据覆盖掉。

由“fifo_empty”信号指示下溢情况，如在“fifo_empty”时不应读取 FIFO，否则会读出垃圾数据。

与握手信号不同，异步 FIFO 用于对性能要求较高的设计中，尤其是时钟延迟比系统资源更为重要的环境中。

如 3.7 节所述，简单的同步 FIFO 可以通过使用双端口 RAM 和单独的读、写端口来实现，读写操作使用同一个时钟。可以用同样的原理设计异步 FIFO，但是要特别注意当产生 FIFO 空和 FIFO 满信号时要避免出现亚稳态现象。

3.8.1 避免用二进制计数器实现指针

以写指针为例。在写请求有效时，写指针在写时钟作用下递增。同样，在读请求有效时读指针在读时钟作用下递增。在产生 FIFO 满信号时，要将写指针与读指针进行比较，由于两个指针与分别与其各自的时钟同步，但是彼此之间异步，在使用二进制计数器实现指针时，就会导致用于比较的指针值取样错误。这在下面说明。

比如，二进制计数器的值会从 FFF 变为 000。这时所有位会同时改变。虽然能通过同步计数器避免亚稳态，但是仍然能得到极不相关的取样值，所以同步计数器不是最终的解决方案。

从 FFF 到 000 可能的转换：

- FFF→000
- FFF→001
- FFF→010
- FFF→011
- FFF→100
- FFF→101
- FFF→110
- FFF→111

如果同步时钟边沿在 FFF 向 000 转换的中间位置到来，就可能将三位二进制数的任何值取样并同步到新的时钟域中。

因为 FIFO 满和 FIFO 空标记的产生要使用这些指针的值，所以错误的指针值将会误产生标记。可能是在 FIFO 满时没有产生 FIFO 满标记，从而使数据丢失；或在 FIFO 空时没有产生 FIFO 空标记，从而读出垃圾数据。

注意

鉴于上面的情况，强烈建议避免使用二进制计数器实现读、写指针。

3.8.2 使用格雷码取代二进制计数

实现 FIFO 指针的一种方式是使用格雷码计数，如表 3.1 所示。

表 3.1 格雷码编码计数器

格雷码/反射码	等效十进制值	格雷码/反射码	等效十进制值
0000	0	1100	8
0001	1	1101	9
0011	2	1111	10
0010	3	1110	11
0110	4	1010	12
0111	5	1011	13
0101	6	1001	14
0100	7	1000	15

格雷码相对于二进制编码的优势在于在从一个数变为另一个数时，只有一位出现变化。

要得到不同的格雷码，只需要从任意一种位组合开始，每次只将其中一位随机从 0 改为 1 或从 1 改为 0，就可以得到其他格雷码。因此格雷码也称为反射码。

因为格雷码是单位间距码，每下一个值与前一个值的区别只有一位距离，所以在转换中最多只会出现一位错误。例如，如果计数器从“1010”变为“1011”，取样逻辑要么读到“1010”（旧值）要么读到“1011”（新值），但是不会出现其他值。

注意

同步格雷码计数器很少会导致取样计数器值出现亚稳态，此外取样后的值最多只有一位出现错误。

同步指针的影响

在 FIFO 满时会阻止进一步访问 FIFO。在 FIFO 满时，按各自时钟递增的读、写指针会进行比较。需要将读指针（格雷码）同步到写时钟上，下面用例子来说明这一点。

如图 3.22 所示，最初在 t_0 阶段读、写指针都为 0。随着后续发生在 FIFO 上的写操作，写指针增加。当到达某个阶段时，读、写指针相等，这时 FIFO 满。在图 3.23 中在 t_5 阶段这种情况发生。

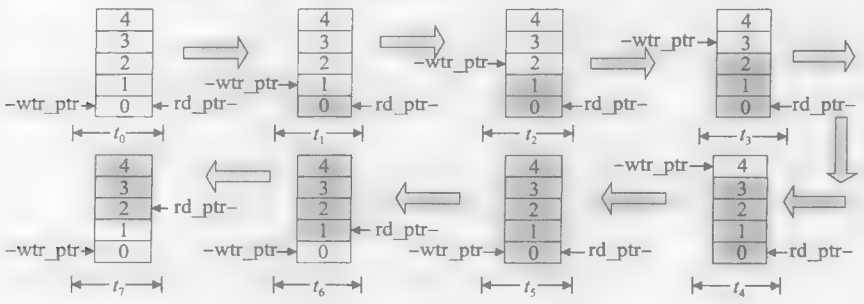


图 3.22 FIFO 满逻辑的同步效果

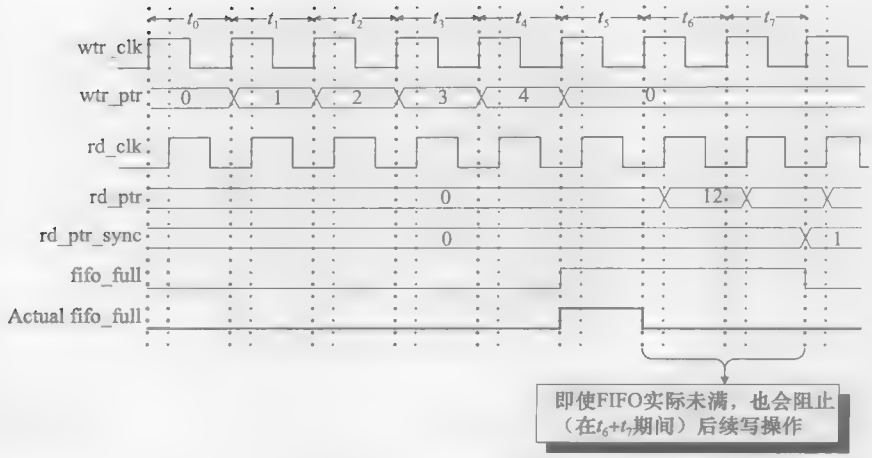


图 3.23 FIFO 满时序

如果在 t_6 处发生了读操作，由于典型的同步电路至少包含两个触发器，将读指针同步到写时钟上将会导致同步后的读指针在两个写时钟后出现。这虽然增加了阻止数据写入的周期，但是对数据的准确性是无害的。只有在 FIFO 实际上已经满了但是没有阻止写动作时才会出现问题。

类似地，在 FIFO 空时也会阻止对 FIFO 进一步的读访问。

对于 FIFO 空计算，把写指针同步到读时钟并与读指针进行比较。因此，在读一边延迟了写操作（延迟了两个时钟信号）并且在实际上有数据时仍然会指示 FIFO 为空。这会导致阻止读操作，直到读一边能看到写操作为止。

如图 3.24 所示，在 t_0 处读、写指针初始值为 0。随着对 FIFO 的写操作，写指针增加。在某个阶段写指针与读指针相等，这时 FIFO 变为满。这在图 3.25 中的 t_3 阶段发生。

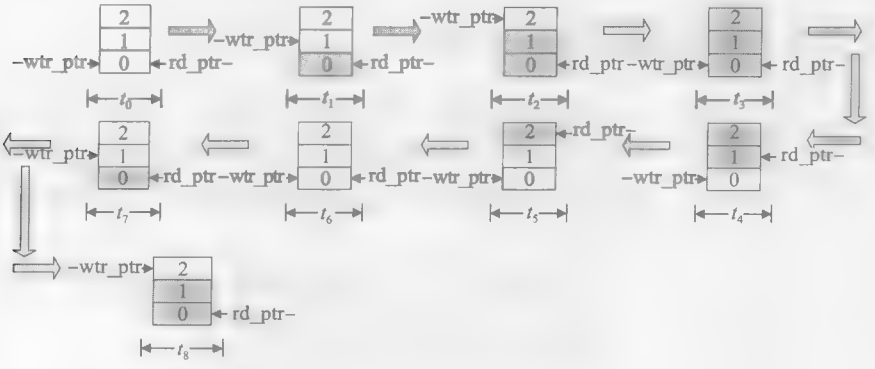


图 3.24 FIFO 空逻辑的同步效果

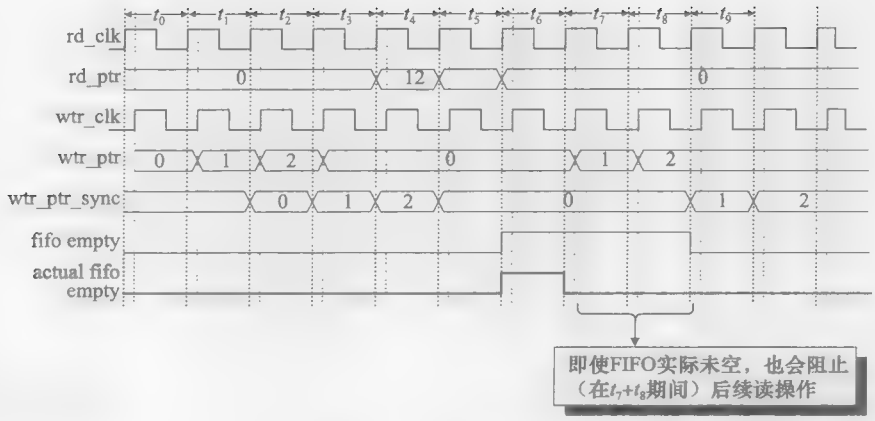


图 3.25 FIFO 空假象

后续的读操作从 t_4 开始，并且 FIFO 在 t_6 再次变为空。在 t_7 和 t_8 又写回 FIFO，由于典型的同步电路由至少两个触发器组成，将写指针同步到读时钟上时将会导致产生最少两个读时钟的延迟。这会导致阻止对 FIFO 额外的读操作，但这是无害的。只有在 FIFO 实际上为空时没有阻止读操作才会产生问题。

注意

在未满时通知写一边 FIFO 已满，或在未空时通知读一边 FIFO 已空都是可以的。即使指针同步后的值（写时同步读指针以及读时同步写指针）会持续一小段时间亚稳态，阻止写/读的影响会使 FIFO 挂起一段时间，但是不会导致任何错误。

3.8.3 用格雷码实现 FIFO 指针

要想完美地产生 FIFO 空或 FIFO 满条件，需要正确取样读、写指针的值。在时钟域间传递指针的最好方式是使用格雷码来实现指针，因为如果同步时钟信号在计数值转换期间到来，这种编码能消除绝大部分的错误。

设计格雷码计数器看起来复杂，但实际上很简单。所有步骤如下：

步骤Ⅰ：将格雷值转换为二进制值。

步骤Ⅱ：根据条件递增二进制值。

步骤Ⅲ：将二进制值转化为格雷码。

步骤Ⅳ：将计数器的最终格雷值保存到寄存器中。

图 3.26 是生成的格雷码计数器。

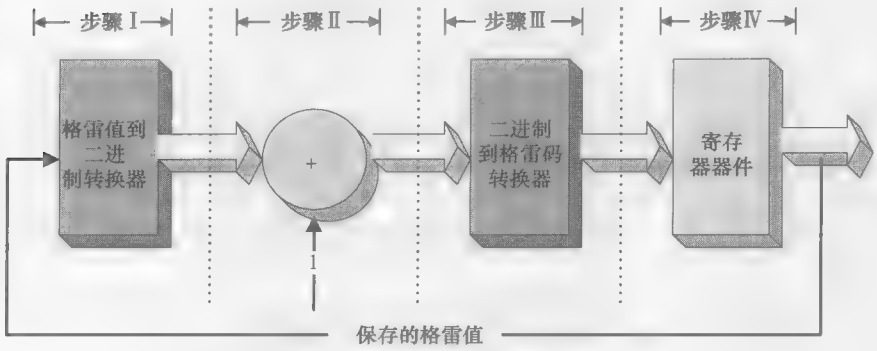


图 3.26 使用二进制加法器的格雷码计数器

3.8.3.1 格雷码到二进制转换器

表 3.2 是分别以格雷码和二进制方式统计的 4 位计数器的值。最后一列是按时钟递增时计数器的转换值。

表 3.2 格雷码/二进制计数器

格雷值	二进制值	等效十进制值
0000	0000	0
0001	0001	1
0011	0010	2
0010	0011	3
0110	0100	4
0111	0101	5
0101	0110	6
0100	0111	7
1100	1000	8
1101	1001	9
1111	1010	10
1110	1011	11
1010	1100	12
1011	1101	13
1001	1110	14
1000	1111	15

格雷码转换为二进制的公式是：

$$\mathrm{bin}_{n-1} = \mathrm{gray}_{n-1} \tag{3.3}$$

$$\text{bin}_i = \text{gray}_i \oplus \text{bin}_{i+1} \tag{3.4}$$

对于 n 位计数器值，$i < n-1$。

图 3.27 为计数器的位编号。

图 3.27 计数器的位编号

让我们举个简单的例子，将格雷码“1010”转换为相等的二进制数。

使 $n-1=3$

将 $i=3$ 代入到式 3.3 中，得到

$$\text{bin}_3 = \text{gray}_3 = \text{gray}[3] = 1$$

将 $i=2$ 代入到式 3.4 中，得到

$$\text{bin}_2 = \text{gray}_2 \oplus \text{bin}_3 = \text{gray}_2 \oplus \text{gray}_3 = \text{gray}[2] \oplus \text{gray}[3] = 1$$

将 $i=1$ 代入到式 3.4 中，得到

$$\text{bin}_1 = \text{gray}_1 \oplus \text{bin}_2 = \text{gray}_1 \oplus \text{gray}_2 \oplus \text{gray}_3 = \text{gray}[1] \oplus \text{gray}[2] \oplus \text{gray}[3] = 0$$

将 $i=0$ 代入到式 3.4 中，得到

$$\text{bin}_0 = \text{gray}_0 \oplus \text{bin}_1 = \text{gray}_0 \oplus \text{gray}_1 \oplus \text{gray}_2 \oplus \text{gray}_3$$
$$= \text{gray}[0] \oplus \text{gray}[1] \oplus \text{gray}[2] \oplus \text{gray}[3] = 0$$

所以我们得到下面 4 个等式

$$\text{bin}[0] = \text{gray}[0] \oplus \text{gray}[1] \oplus \text{gray}[2] \oplus \text{gray}[3] \tag{3.5}$$

$$\text{bin}[1] = \text{gray}[1] \oplus \text{gray}[2] \oplus \text{gray}[3] \tag{3.6}$$

$$\text{bin}[2] = \text{gray}[2] \oplus \text{gray}[3] \tag{3.7}$$

$$\text{bin}[3] = \text{gray}[3] \tag{3.8}$$

基于以上的等式，格雷码“1010”对应的二进制数为“1100”。

从上面的等式可以看出，bin[3] 可以通过将格雷值右移 3 位得到。bin[2] 可以通过将格雷值右移 2 位得到，bin[1] 可以通过将格雷值右移 1 位得到，bin[0] 可以通过将格雷值右移 0 位得到[43]。

下面是上面的格雷码到二进制码转换器的 Verilog 代码。

```
module gray_to_bin (bin , gray);
  parameter SIZE = 4;
  input [SIZE - 1:10] bin;
  output [SIZE - 1:10] gray;
  reg [SIZE - 1:10] bin;
  integer i;
always @ (gray)
  for ( i = 0; i <= SIZE; i = i + 1)
   bin[i] = ^(gray >> i);
endmodule
```

3.8.3.2 二进制－格雷码转换器

下面的等式用于二进制向格雷码的转换：

$$\text{gray}_{n-1} = \text{bin}_{n-1} \tag{3.9}$$

$$\text{gray}_i = \text{bin}_i \oplus \text{bin}_{i+1}, \text{这里 } i < n-1 \tag{3.10}$$

我们来看一个将二进制码“1100”转换到对应格雷码的简单例子。

取 $n-1=3$

将式（3.9）中的 i 值取 3，得到

$$\text{gray}_3 = \text{bin}_3 = \text{bin}[3] = 1$$

将式（3.10）中的 i 值取 2，得到：

$$\text{gray}_2 = \text{bin}_2 \oplus \text{bin}_3 = \text{bin}[2] \oplus \text{bin}[3] = 0$$

将式（3.10）中的 i 值取 1，得到：

$$\text{gray}_1 = \text{bin}_1 \oplus \text{bin}_2 = \text{bin}[1] \oplus \text{bin}[2] = 1$$

将式（3.10）中的 i 值取 0，得到：

$$\text{gray}_0 = \text{bin}_0 \oplus \text{bin}_1 = \text{bin}[0] \oplus \text{bin}[1] = 0$$

这样就得到了已知二进制码“1100”所对应的格雷码“1010”。

基于上例，我们得到以下 4 个等式：

$$\text{gray}[0] = \text{bin}[0] \oplus \text{bin}[1] \tag{3.11}$$

$$\text{gray}[1] = \text{bin}[1] \oplus \text{bin}[2] \tag{3.12}$$

$$\text{gray}[2] = \text{bin}[2] \oplus \text{bin}[3] \tag{3.13}$$

$$\text{gray}[3] = \text{bin}[3] \tag{3.12}$$

从式（3.11）～（3.14）可以看出，可以通过逐位异或，或者将二进制码右移后与自身异或操作的方式，计算出对应的格雷码，如下所示：

```
bin[3]   bin[2]   bin[1]   bin[0]   → 二进制值：二进制
   0     bin[3]   bin[2]   bin[1]   → 右移值(二进制)
-------------------------------------------------
gray[3]  gray[2]  gray[1]  gray[0]  → 对应的格雷码值
```

下面是上述二进制-格雷码转换器的 Verilog 代码：

```
module bin_to_gray (bin, gray);
  parameter SIZE = 4;
  input [SIZE-1:0] bin;
  output [SIZE-1:10] gray;
  assign gray = (bin >> 1) ^ bin;
endmodule
```

3.8.3.3 格雷码计数器的实现

将图 3.26 中的所有四个步骤组合在一起（格雷码到二进制码转换

器、加法器，二进制码到格雷码转换器以及最后用于保存格雷值的寄存器组）。

下面是格雷码计数器的 Verilog 代码。

```
module gray_ counter (clk, gray, inr, reset_n)
  parameter SIZE = 4;
  input clk, inr, reset_n;t
  output [SIZE -1 ] gray;
  reg [SIZE] - 1 ] gray_temp, gray, bin_temp, bin;
  integer i;

always @ (gray or inr)
begin:gray_bin_gray
  for (i = 0; i<SIZE ; 1 = i +1)
   bin[i] = ^(gray >> i);           // gray to binary conversion
  bin_temp = bin + inr;             // addition in binary
  gray_temp = bin_temp >> 1) ^ bin_temp; // binary to gray conversion
end

endmodule
```

下面的 always 块将转换后的格雷值寄存起来：

```
always @ (posedge clk or negedge reset_n)
begin:gray_registered
if (~reset_n)
  gray <= {SIZE {1'b0}};
else
  gray <= gray_temp;
end
```

图 3.28 是以上格雷码计数器的逻辑原理图。

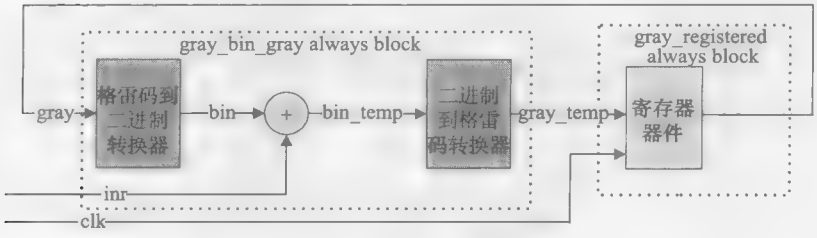


图 3.28 格雷码计数器逻辑

3.8.4 FIFO 满和 FIFO 空的产生

N 位指针可以覆盖 FIFO 中的 2^N 个地址。在两个指针相等时，因为 FIFO 可能处于空状态也可能处于满状态，所以需要使用额外的位来对这两

种情况进行区分。

当二进制码指针的最高有效位不同，而其余位相同时，FIFO 为满。

当二进制码指针的所有位都相同时，FIFO 为空。下面用一个例子加以说明：

例如，一个深度为 8 的 FIFO。使用 3 位表示所有 8 个位置，并用额外的一位来区分 FIFO 满或空。一开始 rd_ptr_bin 和 wr_ptr_bin 都是“0000”，FIFO 为空。在连续 8 次写 FIFO 后得到读、写指针的值为：

$$rd\_ptr\_bin = "0000"$$

$$wr\_ptr\_bin = "1000"$$

这就是图 3.29 中所示的 FIFO 满状态[45]。

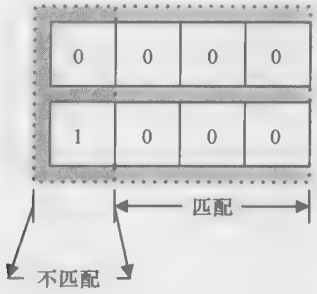


图 3.29 FIFO 满条件

现在连续读 8 次，得到以下读、写指针值：

$$rd\_ptr\_bin = "1000"$$

$$wr\_ptr\_bin = "1000"$$

这就是图 3.30 中所示的 FIFO 空状态。

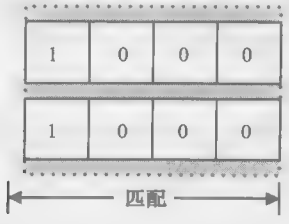


图 3.30 FIFO 空条件

图 3.31 是 FIFO 空和 FIFO 满的模块原理图。

在这种情况下，由于存在 XOR 门链路，最大操作频率取决于格雷码

计数器的速度。

由于读/写指针的值以格雷码保存，而所有的比较和递增操作以二进制码形式进行，使得实现和纠错变得相当简单。在图3.31中，需要4个格雷码到二进制码转换器，如果直接用格雷码实现计算FIFO空和FIFO满进行的比较等操作就可以避免使用这些转换器。这会增加复杂度并需要额外的逻辑。我们将在下一节中看看这种方法是如何工作的。

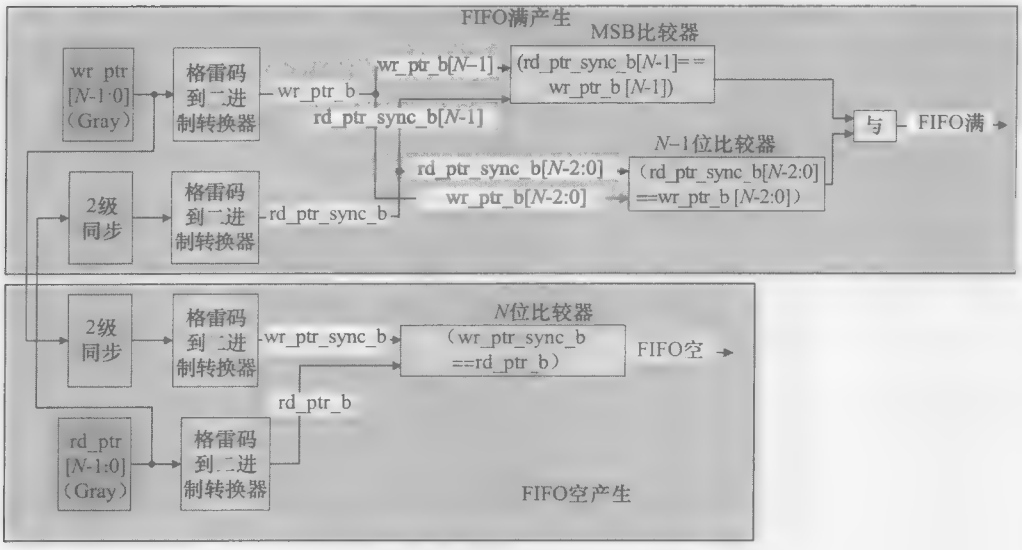


图3.31 FIFO满和空信号产生逻辑

产生FIFO满和FIFO空的另一种方法

该方法要求创建两个格雷码计数器，一个n位，另一个$n-1$位。可以先创建一个n位格雷码计数器，然后通过修改其第二个MSB来产生一个$n-1$位的格雷码计数器，这两个计数器的LSB相同[3]。

在我们开始主逻辑之前，先详细了解格雷码计数器。

图3.32是4位格雷码计数器。

如图3.32所示，除MSB之外的所有列中的位相对于中线是对称的。后半段格雷码是前半段格雷码MSB反转后的镜像。

现在，就可以将n位格雷码中2个MSB进行异或操作后的值作为$n-1$位格雷码的MSB。其余的$n-2$位可以直接用n位格雷码的$n-2$位替代。图3.33显示了4位到3位格雷码的转换（见表3.3）。

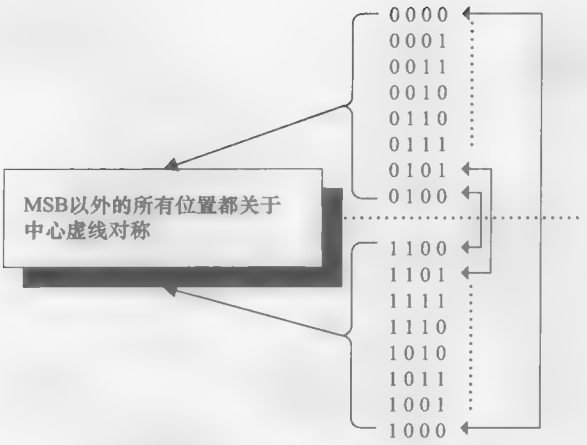


图 3.32 4 位格雷码计数器

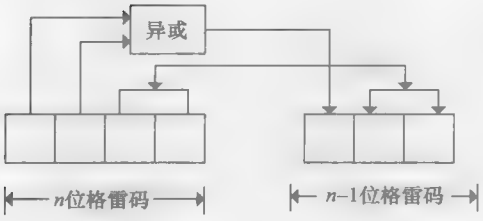


图 3.33 4 位格雷码向 3 位格雷码的转换

表 3.3 4 位格雷码向 3 位格雷码的转换

4 位格雷码	转换后的 3 位格雷码	4 位格雷码	转换后的 3 位格雷码
0000	000	1100	000
0001	001	1101	001
0011	011	1111	011
0010	010	1110	010
0110	110	1010	110
0111	111	1011	111
0101	101	1001	101
0100	100	1000	100

在下一节的 FIFO 设计中，会介绍如何将双 n 位格雷码计数器用于 FIFO 空/满产生逻辑。

3.8.5 双时钟 FIFO 设计

图 3.34 中是使用双端口存储器作为存储器件的 FIFO 模块图。

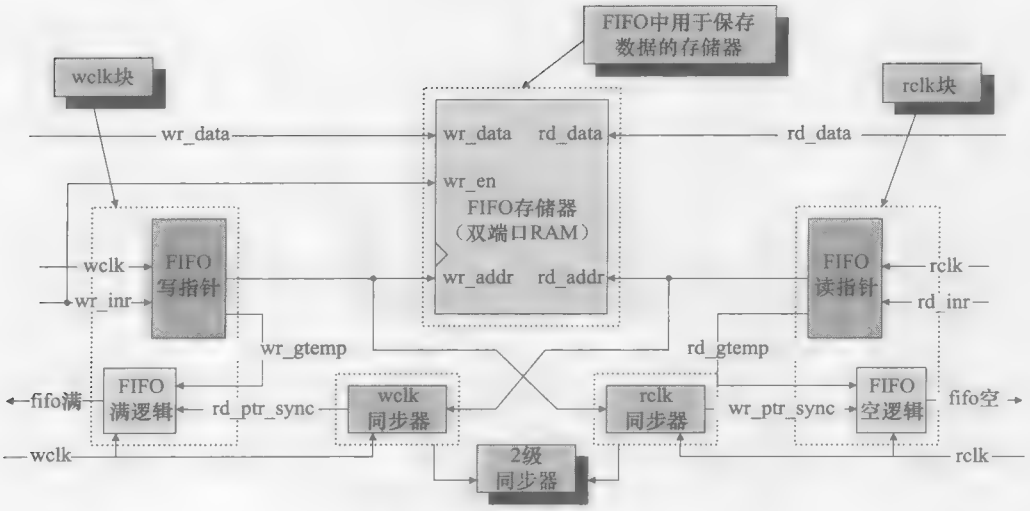


图 3.34 双时钟 FIFO 设计

3.8.5.1 FIFO 空条件的产生

当读指针与同步后的写指针相匹配时，FIFO 为空，这时应该在 FIFO 的读时钟域内马上产生 FIFO 空标记。

注意与 3.8.4 节中所示的实现方式不同，这时比较读和同步化的写指针的格雷码。如果在比较前指针首先转化为自身的等效二进制码，这种方式就能节省所需要的 4 个格雷码到二进制码转换器。

与之前的实现类似，指针宽度比寻址 FIFO 存储器所需的宽度多一位。同步后的写指针（wr_ptr_sync）与 rd_gtemp 进行比较（下一个格雷码将保存在 rd_ptr 内）。

下面是以上逻辑的 Verilog 代码。

```
always @ (posedge rclk or negedge reset_n)
 begin: fifo_empty_gen
  if (~reset_n)
     fifo_empty <= 1'b1;
  else
     fifo_empty <= (rd_gtemp = = wr_ptr_sync);
 end
```

注意

生成的 FIFO 空输出寄存起来。

3.8.5.2 FIFO 满条件的产生

当写指针与同步化的读指针相等时，FIFO 为满，这时应该在 FIFO 的

写时钟域内马上产生 FIFO 满标记。

注意，应直接比较写指针和同步化的读指针的格雷码。

与之前的实现类似，指针宽度比寻址 FIFO 存储器地址所需的宽度多一位。由于这次直接使用格雷码而不是二进制码对指针进行比较，因此产生该条件的逻辑与之前的实现方式不同。

让我们用一个例子来做以说明。

图 3.35 中是深度为 8 的 FIFO 执行的一系列步骤。

步骤 1：FIFO 初值为空，即“rd_ptr” = “wr_ptr” = 0，如图 3.35 所示。

步骤 2：连续写 FIFO，直到 FIFO 满，此时“rd_ptr” = 0 且“wr_ptr” = 7。现在连续读 FIFO 至“rd_ptr” = “wr_ptr” = 7，这时 FIFO 变空（因为读和写指针的所有位相等），如图 3.35 所示。

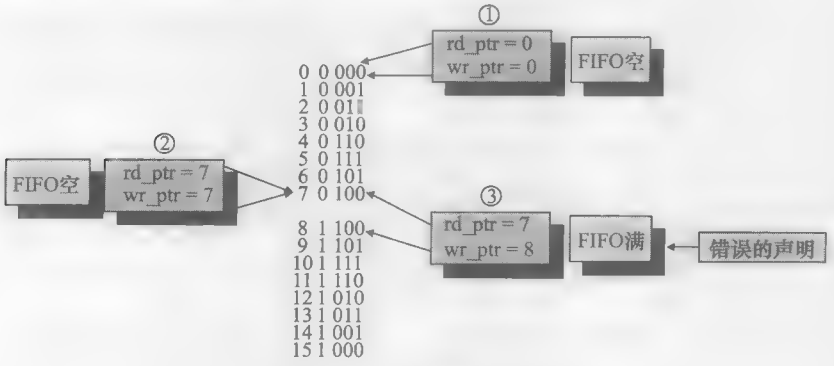


图 3.35 FIFO 满和空条件

步骤 3：这时再进行一次写操作，使“rd_ptr” = 7 且“wr_ptr” = 8。如果使用之前的二进制码比较逻辑（见 3.8.4 节），FIFO 会再次指示满，即使它并没有满（见图 3.36）。

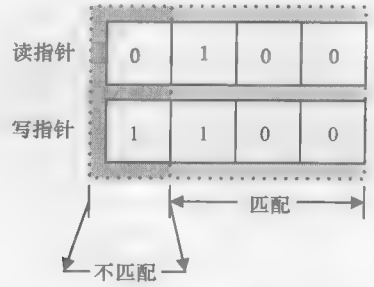


图 3.36 FIFO 满条件

使用二元 n 位格雷码计数器能很容易地处理这种状况。

正确地进行满比较的方式是将“rd_ptr”同步到写时钟域内。然后比较 MSB，并在写指针比读指针多回绕一次时作以区分。如果同步后读指针的 MSB 为高，同步后读指针（rd_ptr_sync）的第二个 MSB 在与 $n-1$ 位写指针进行比较前反转。

所以在以下三个条件都为真时，FIFO 满标志置起。

1）同步后的读指针（rd_ptr_sync）的 MSB 应该与写指针（wr_gtemp）的下一个格雷码值的 MSB 不同，wr_gtemp 将寄存到 wr_ptr 中。

2）写时钟域中下一个格雷码计数值的第二个 MSB（wr_gtemp），应该与同步到写时钟域内读指针的第二个 MSB 相同（rd_ptr_sync）。

3）两个指针中所有省略掉的 LSB 都应该匹配。

注意

上面第 2 个要点中的第二个 MSB 通过将指针前两个 MSB 异或后计算出来。（如果 MSB 为高，对两个 MSB 进行异或操作会使第二个 MSB 取反。）

下面是以上逻辑的 Verilog 代码。

```
wire rd_2nd _msb = rd_ptr_sync [SIZE] ^ rd_ptr_sync [SIZE - 1];
wire wr_2nd_msb = wr_gtemp [SIZE] ^ wr_gtemp [SIZE-1];
always @ (posedge wclk or negedge reset_n)
  begin: fifo_full_gen
   if (~reset_n)
    fifo_full <= 1'b0;
   else
    fifo_full <= ((wr_gtemp [SIZE] != rd_ptr_sync[SIZE]) &&
            (rd_2nd_msb = = wr_2nd_msb) &&
            (wr_gtemp[SIZE -2:0] = = rd_ptr_sync[SIZE
            -2:0]));
end
```

参考文献

1. Arora M, Bhargava P, Gupta S (2002) Handling multiple clocks (problems & remedies in designs involving multiple clocks). DCM Technologies, SNUG India
2. Cummings CE (2001) Synthesis and scripting techniques for designing multi-asynchronous clock designs. In: SNUG 2001 (Synopsys Users Group Conference), San Jose. User Papers

第4章

时钟分频器

4.1 介绍

典型情况下 SoC 要对设计中各种组件提供许多与相位相关的时钟。将主时钟以 2 为幂次进行分割来产生同步偶数分频时钟。然而，有时也会需要按奇数甚至小数进行分频。在这些情况下，如果没有更高频的主时钟，无法得到同步分频时钟。

虽然偶数分频时钟始终产生 50% 占空比的输出，但有时在奇数或小数分频时也需要产生 50% 占空比的时钟。本章提供了实现这些不常见时钟分频器的指南和细节。本章首先介绍简单的奇数分频器（按 3、5 等分频），然后扩展到非整数分频器（按 1.5、2.5 等分频）。所描述的电路比选用任何外部 PLL 都更为简单有效、便宜和快速。

4.2 同步整数分频器

同步整数分频可以用 Moore 状态机很容易地实现。

图 4.1 使用 Moore 状态器进行 7 分频。

但是这样简单的逻辑无法产生 50% 占空比的输出。

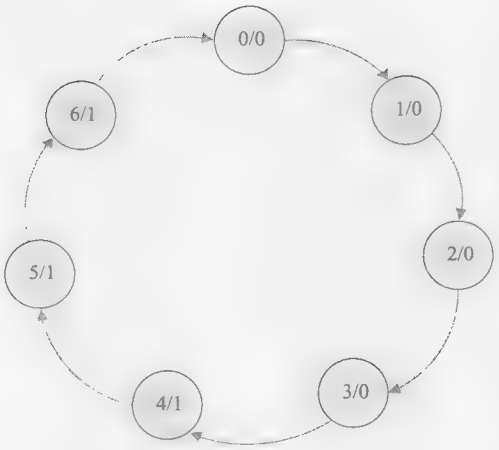


图4.1 7分频 Moore 状态机

4.3 具有50%占空比的奇数整数分频

在概念上，产生具有50%占空比的奇数分频时钟最简单的方式是以期望输出频率的一半生成两个正交相位时钟（两个时钟之间有90°相位差）。

然后通过将两个波形异或得到输出频率。

由于存在固定的90°相位差，每次异或输入只有一端会变化，这样有效消除了输出波形上的毛刺。

让我们来看看一个3分频的简单例子。

下面列出整奇数分频的几个步骤[96]。

步骤Ⅰ：创建由时钟上升沿触发的0到（$N-1$）的计数器，N是自然数，用于对参数时钟进行分频（N不等于偶数）

对3分频，从0计数到2…$N=3$

对5分频：从0计数到4…$N=5$

对7分频：从0计数到6…$N=7$

注意

计数器在输入时钟（ref_clk）上升沿累加，在到达（$N-1$）后自动归零。

步骤Ⅱ：使用两个开关触发器，按以下方式产生其使能信号。

tff1_en：TFF1在计数值为0时使能。

tff2_en：TFF2在计数值为（$N+1$）/2时使能（2对应3分频，3对应5分频，依此类推），如图4.2所示。

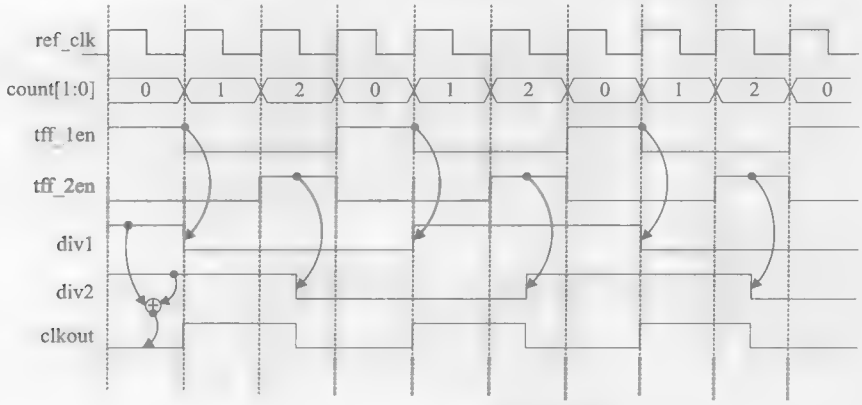


图4.2 具有50%占空比的3分频时序图（$N=3$）

步骤Ⅲ：产生以下信号。

div1：TFF1的输出→由输入时钟（ref_clk）上沿触发。

div2：TFF2的输出→由输入时钟（ref_clk）下沿触发。

注意

两个T触发器的输出“div1”和“div2”产生2N分频后的波形，如图4.2所示。

步骤Ⅳ：通过异或“div1”和“div2”波形，产生最终输出时钟“clkout”（N分频）

图4.3为该3分频电路的逻辑结构图。

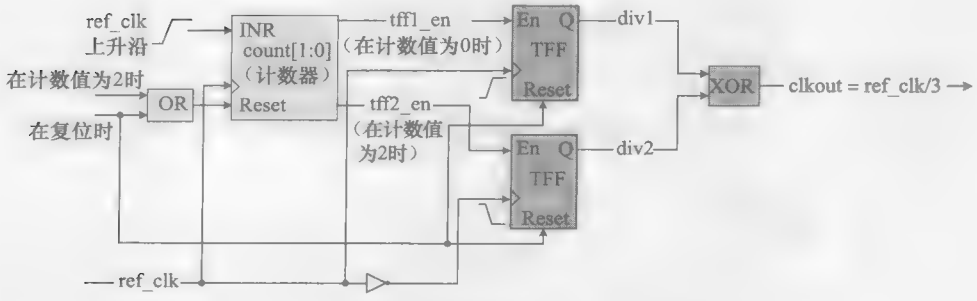


图4.3 用T触发器进行50%占空比3分频

4.4 非整数分频（非50%占分比）

经常会使用 N 分频电路来产生与参考时钟不同步的时钟。设计 N 为非整数的分频电路并不像看起来那样难。让我们来看一下 1.5 分频的含意。简单来说，就是每三个参考时钟包含两个对称的脉冲。

下一节介绍一个时钟按 1.5 分频的非 50% 占空比的例子。

4.4.1 具有非50%占空比的1.5倍分频

图 4.4 中的多路器在“clkout”为高时选择 ref_clk，否则它选择反向后的 ref_clk。

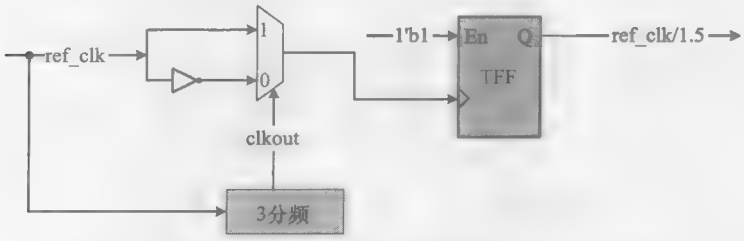


图 4.4 使用T触发器进行 1.5 倍分频（非 50% 占空比）

图 4.5 是图 4.4 中所示的 1.5 倍分频电路的时序图。

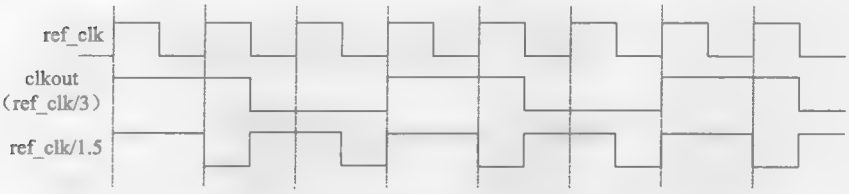


图 4.5 使用开关寄存器进行 1.5 分频的时序图（非 50% 占空比）

注意

上述电路在仿真时可以正确工作，但是在综合时可能出现问题，因为在多路器的选择端切换开关时两个输入端的延时并不相等。多路器的输出不能马上改变并可能在输出时钟上产生毛刺。随着参考时钟（ref_clk）频率的增加，出现错误的可能性会越来越大。

4.4.2 4.5倍分频计数器的实现（非50%占空比）

本节介绍另一种方法，可以对非整数分频电路进行优化，使其输出时钟完全不含毛刺。

让我们使用4.5倍分频的例子，即每9个参考时钟包含2个对称脉冲。

下面是非整数分频的一系列步骤。

步骤Ⅰ：使用复位值为000000001的9位移位寄存器，可以在每个时钟上升沿使移位寄存器循环左移一位。

步骤Ⅱ：要产生第1个脉冲，必须使在半周期时移动第1位并将第1位与第2位进行或操作。

步骤Ⅲ：要产生第2个脉冲，第5位和第6位必须在半周期时移动并与原始第6位进行或操作。

注意

所有这些移位都是用来保证输出波形不含毛刺的必要操作。

上面产生的时钟占空比为40%，并且输出不含毛刺。

图4.6是4.5分频电路的时序图，下面是该逻辑的Verilog代码。

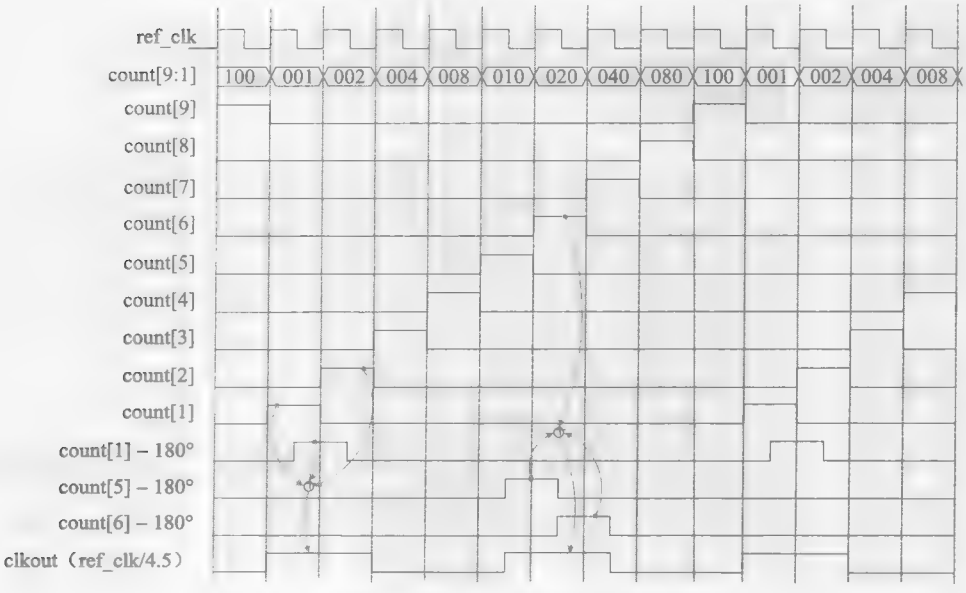


图4.6 4.5倍分频计数器的实现（非50%占空比）

```
/* Counter implementation
   reset value : 9'b000000001 */
  always @ ( posedge ref_clk or negedge p_n_reset)
    if (!p_n_reset)
      count [9:1] <= 9'b000000001;
    else
      count [9:1] <= count [9:1] << 1;
  /* count bit 1st, 5th and 6th phase shifter by 180 deg */
   always @ (negedge ref_clk or negedge p_n_reset)
     if (!p_n_reset)
       begin
         ps_conunt1 <= 1'b0;
         ps_count5 <= 1'b0;
         ps_count6 <= 1'b0;
       end
     else
       begin
         ps_count1 <= count[1];
         ps_count5 <= count[5];
         ps_count6 <= count[6];
       end
// Genration of final output clock = (ref_clk / 4.5)
     assign clkout = (ps_count 5 | ps_count 6 | count [6]) |
                     (count [0] | count [1] | ps_count1);
```

4.5 N分频的替换方法

上面各电路都假设输入时钟为50%占空比，否则小数分频器输出会出现抖动，而且整数分频器也会有不同的占空比。

所有使用基于LUT（查找表）的组合反馈的电路都能工作得很完美，并产生不含毛刺的时钟。

让我们再次回到简单的1.5分频电路。

1.5倍分频的查找表实现

通过图4.7所示的3分频电路产生1.5倍分频。在图4.7中两个触发器组成了3分频电路（非50%占空比）[96]。

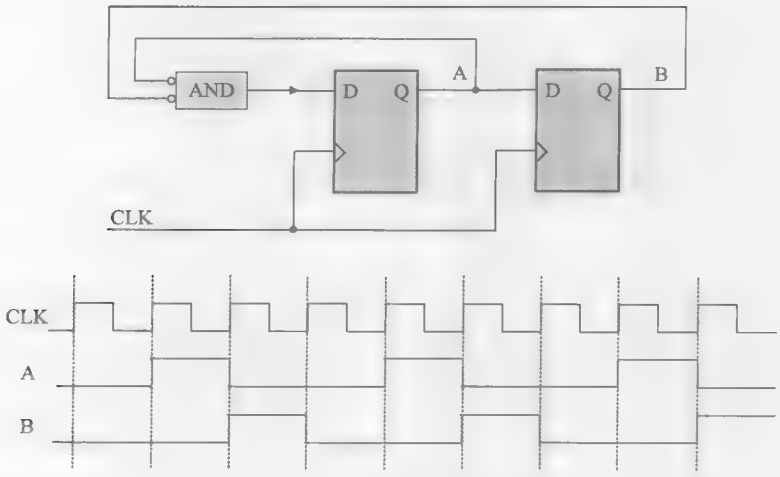


图 4.7　1.5 倍分频（非 50% 占空比）

参考文献

1. Arora M (2002) Clock dividers made easy, ST Microelectronics, SNUG Boston

第5章

低功耗设计

5.1 介绍

在早期的IC设计中，功耗并不是一个重要的约束条件，所以在设计时往往不对其作以考虑。随着使用电池供电的器件变得更小，功能更多，对功耗的要求也越来越高。

虽然对于在使用电池的应用中，如起搏器和数字电子表，低功耗设计技术已经使用了很多年，但如今的技术趋势使得这些技术得以更广泛地使用。

能量以热量形式消耗。对于设备制造商，发热管理仍然是一个主要的考虑因素。可靠性是温度的函数，据估计温度每升高10°C失效率可能提高一倍。保持低温操作环境意味着使用散热片或风扇来散热——这会增加整体的重量和成本。如果能在SoC级对功耗进行控制，就可以减少甚至可能消除掉这些开支，也就可以得到更小、更便宜和更可靠的最终产品。

近来对于能源效率目标的关注达到了前所未有的高度。本章描述了减少动态和静态功耗的各种技术。

5.2 功耗源

浪涌、静态功耗和动态功耗是三个主要的功耗源。浪涌电流指器件上电时产生的最大瞬时输入电流。浪涌电流在应用中也称为启动电流。

浪涌电流与设备相关。如电机的启动电流在前几个周期中是正常满载电流的数倍。

基于 SRAM 的 FPGA 也有很突出的浪涌电流，因为上电时这些器件没有配置，所以需要从外部存储器中下载数据来配置其编程资源，如布线连接和查找表。反之，反熔丝 FPGA 由于无须上电配置，所以没有浪涌电流。

待机电流指在关断主电源或系统进入待机模式下产生的电流。由待机电流产生的功耗称为待机功耗。与浪涌功耗极为类似，待机功耗与元件的电气特征密切相关。待机功耗也称为静态功耗。需要指出的是，静态功耗也包含电路中由晶体管的漏电流所导致的功耗。

动态功耗或开关功耗是门电路输出切换时，由逻辑转换所引起的功耗。

功态功耗（$P_{dynamic}$）由下面的等式定义：

$$P_{dynamic} = SC_L V_{dd^2} f_{clk}$$

式中 C_L = 门寄生电容

S = 每个时钟通过整个电路的平均转换次数

f_{clk} = 时钟频率

V_{dd} = 供电电压

输出从逻辑 0 变为逻辑 1，电容充电，从逻辑 1 到逻辑 0，电容放电。注意，开关功耗也是时钟频率 f_{clk} 和供电电压 V_{dd^2} 的函数。

因此，ASIC 的总功耗定义为：

$$P_{total} = P_{dynamic} + P_{static}$$

对若干时钟周期使用这些方程，就得到了平均功耗。

功态功耗在大规模 IC 设计中占主要地位。在典型的应用中动态功耗占到了总功耗的 80%。

5.3 在各设计抽象层次降低功耗

降低功耗应当在所有设计层次上进行。即，在系统级、逻辑级和物理级。层次越高对功耗降低就可能越有效。

图 5.1 展示了在不同设计层次上降低功耗的各种设计技术。虽然功耗可以在各设计层次降低，但是最好在更高的抽象层次上进行，即，在系统和体系结构级可以达到最大的降低效果。

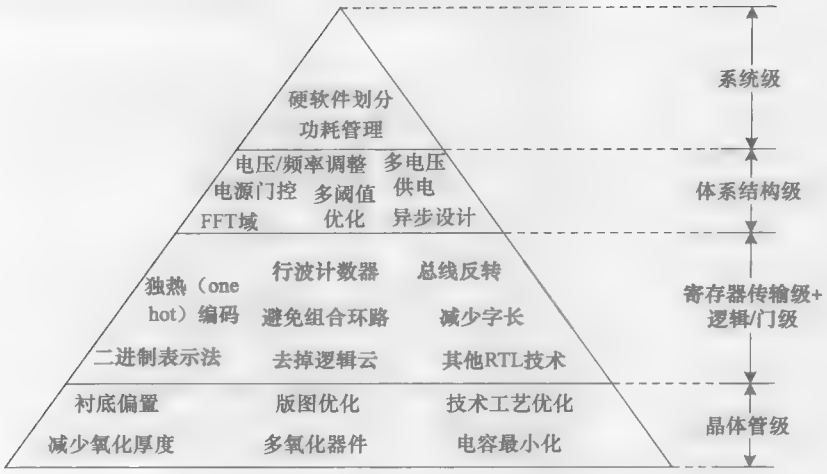


图 5.1 各抽象层次降功耗图

为了最大程度降低功耗，各抽象层次在设计时就要把功耗因素考虑在内。

再次回到前面对整体功耗描述的方程上，可以看出可以通过降低电压、电容、信号频率和单元能耗来降低整体功耗。在特定的层次上，指定技术会涉及这些因素中的一个或多个。

许多系统级的设计决定与应用密切相关，即，是选择基于缓存的存储器还是集中式存储器。在体系结构级往往需要选择使用并行结构还是流水线结构，并行结构比多路复用的方式能效更高。在逻辑和版图级，对映射网表方法的选择和低功耗库的选择是关键。在物理级，要使用版图优化技术。

表 5.1 展示了各抽象设计层次中对功耗降低程度的影响。

表 5.1 不同层次的功耗降低比率

抽象级	功耗降低几率
系统级	10% ~100%
体系结构级	10% ~90%
寄存器传输级	15% ~50%
逻辑/门级	15% ~20%
晶体管级	2% ~10%

下面几节介绍了在从系统到晶体管级之间的各层次上所使用的各种低功耗技术。

5.4 系统级低功耗技术

在系统设计之前，必须对系统及其性能以及功耗的目标作以考虑。

5.4.1 片上系统方法

对于纳米级高端芯片，由于I/O使用比芯片内核逻辑更高的电压供电（典型值为3.3V），使得其占到总功耗的50%以上。如果整个系统包含多块芯片，这些芯片间的连线将消耗大量的功耗。在现代数字设计实践中，片上系统方法学主要关注降低功耗，缩减面积以及降低成本的手段。

5.4.2 硬件/软件划分

由于嵌入式处理器在大规模数字系统中广泛使用，因此某些功能可以用硬件实现，其余部分可以用软件实现。

通信算法具有高度递归的性质，这意味着少量的代码就可以负责大量的处理任务。事实上10%的代码花费了90%的执行时间。如果这些资源密集型模块能用硬件标识和实现，就能节约大量功耗。这种递归模块可能只占整体系统很小的一部分，但能显著地降低功耗。

文献［10］的作者发布了色键算法，用22 000行代码实现。其中只有15行关键的循环部分用硬件实现，但降低了77%的功耗。

协同设计的常规技术是在设计的早期将系统划分为硬件和软件部分并反复优化以得到最佳方案。这种方法代价较高并比较费时。

典型的设计流程在文献［11］中进行了介绍：

- 性能规范
- 划分
- 综合
- 集成
- 协同设计
- 验证

图5.2为硬软件协同设计的常规方法[11]。

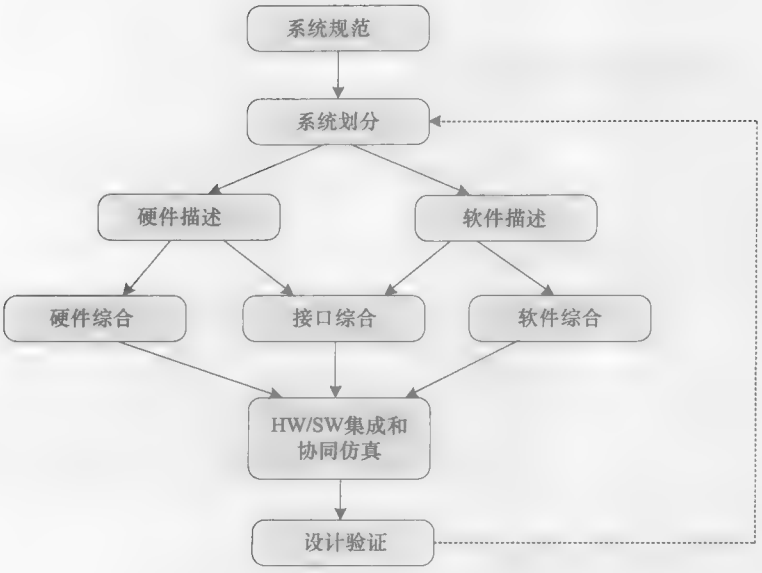


图 5.2 硬/软件协同设计的常规方法

系统的设计过程从性能规范开始。系统设计者根据规范和自身经验对系统性能作出推测。根据推测来决定系统哪些部分用硬件实现（以 ASIC 的形式），哪些部分用软件实现。

这也涉及对系统不同部分的行为进行描述。例如，硬件部分可以用 VHDL 或 Verilog 语言进行描述，软件部分用 C 语言进描述。此外，接口逻辑（包括任意信号交换或总线逻辑）也需要明确下来。

可以使用不同的工具从行为模式中提取出物理模型，比如，使用硬件综合工具从用 Verilog 或 VHDL 描述的模块中提取出物理模型。与之类似，编译器也可以将用高级语言编写的程序编译为嵌入式处理器所使用的原生指令集。

下一步是使用商业级协同仿真平台对在集成环境下已综合的硬件和软件模型进行协同仿真。协同仿真用于验证设计功能的要求和规范的约束。如果系统不满足要求，整个过程要从系统划分开始重新再来。

一种比常规方式更有效的 HW/SW 划分方法是使用基于模型的方式，如图 5.3 所示。

这是一种基于给定规范建立系统模型的想法。模型要么完全自己建立要么使用可复用的已有模型库。随着库的增加，可以大大缩短设计时间。

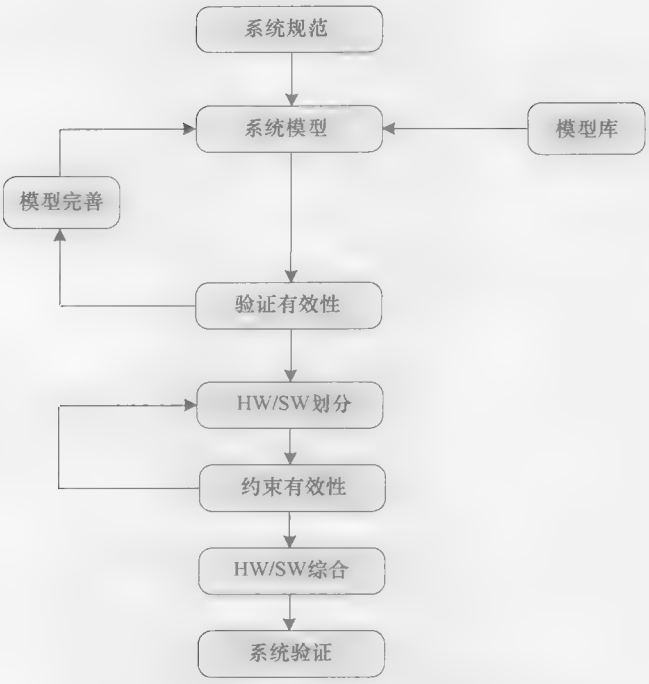


图 5.3 用基于模型的方法进行 HW/SW 协同设计

可以使用 SystemC 建立模型，这种语言是 C++ 语言的一组系统扩展库，用于对硬件建模。SystemC 程序可以用标准 C++ 编译器编译并产生用于仿真的目标代码。

SystemC 可以很灵活地创建精确到周期的高抽象级模型来描述整个系统。

仿真的结果用来验证模型。将仿真输出与期望值进行比较。验证的结果也可以用来完善模型。

系统级建模和 HW/SW 划分的详细说明不在本书的描述范围之内。

5.4.3 低功耗软件

硬件设计人员在设计集成芯片或 ASIC 时往往会考虑功耗因素。但大多数软件工程师并不会这样做。另外，通过修改应用软件可以大量降低功耗，得到“更加绿色”和能效更高的系统。

高级语言便于使用并可以事半功倍。然而，有些结构难以用其实现，而且实现高级语言的运行时环境使用高频轮询来实现这样的结构，这会导

致较高的能耗。所以在使用高级语言时要避免使用复杂原语。

对于嵌入式应用，常常在设计中使用现有的工业级 C 代码。C 代码可能会使用若干循环。在一些应用中，90% 的运时时间可能都在执行这些循环。

可以使用几种技术来优化这些循环。如果两个循环在同样的索引下逐一执行，可以将它们合并。执行的指令数由此减少。例如，按下面所示的方式将循环合并后，由于移除了循环计数器（初始化、递增和比较），使循环指令数目减少了。

其他的考虑可能包括基于确定硬件体系结构或处理器指令和寄存器的实现。需要注意的是，基于 uP 的实现会产生高序列，因为内部寄存器（ALU 加法器）可以用来快速地进行这些操作。例如，可能只需一步就可以更新硬件计数器，而在后面将提到在软件中实现同样的操作需要许多时钟周期来顺序执行若干条指令。

5.4.4 选择处理器

处理器的选择会对整体功耗产生明显影响。第一点是采用适合所要求的数据宽度的处理器。使用 8 位微控制器来处理 16 位数据将会增加大量排序。对 16 位乘法，使用 16 位处理器需要 30 条指令（加 - 移位算法），而在 8 位器机上则需要 127 条指令（双精度）。更好的结构是使乘加单元（MAC）或 16×16 位并行乘法器用一条指令来执行乘法。

如果用简单的 MAC 就能满足运算要求，就没有必要使用专用 DSP 处理器来处理数据，这样能显著降低功耗。

图 5.4 中是一个有趣的节省功耗的系统体系结构[12]。对于任何应用，都利用微处理器进行控制，同时用协处理器或 DSP 进行数据处理。最好的系统体系结构是使用特定的机器（协处理器）来执行这个任务，以使任务在最小且能效最高的机器中完成。大多数情况下，微控制器和协处理器不会并行运行。

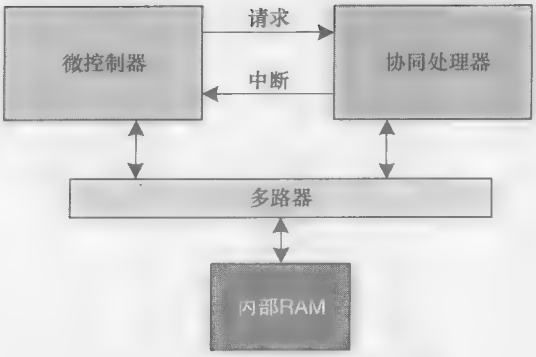


图 5.4 微控制器和协同处理器

5.5 体系结构级降低功耗技术

系统使用不同的体系结构对于功耗会有显著的影响。本节介绍一些体系结构级的权衡和技术以减少功耗。

5.5.1 高级门控时钟

在同步数字系统中，时钟分布贡献了整个数字开关功率中的绝大部分。在许多情况可以通过门控时钟将绝大部分不使用的电路关闭掉。

组合门控时钟已经在 2.5 节中介绍过。组合门控时钟基于从寄存器的输出到输入之间存在着反馈回路这样的事实，所以也称为“基于反馈回路的门控时钟”。注意，在组合门控时钟中，在插入门控时钟之前和插入之后电路的功能并没有改变，所以可以用一致性检查工具进行验证。在本节中我们会接触到一些高级的门控时钟技术。

因为组合门控时钟方案在输出不变时使触发器的时钟失效，所以它可以降低 5% ~10% 的动态功耗。反之，时序门控时钟在不影响设计功能的情况下改变设计结构。时序门控时钟能减少连接到带有门控时钟的寄存器块的设计部分的冗余切换。

图 5.5 为组合门控时钟示意图，图 5.6 为同样电路的时序门控时钟示意图。注意，在使用时序门控时钟时，后序的流水线阶段也使用同样的条件进行门控操作。

在图 5.6 中可以看到时序门控时钟在实现会加入额外的逻辑。因为需

要这样多余的逻辑，所以该技术并不适用于多位宽数据的情况。

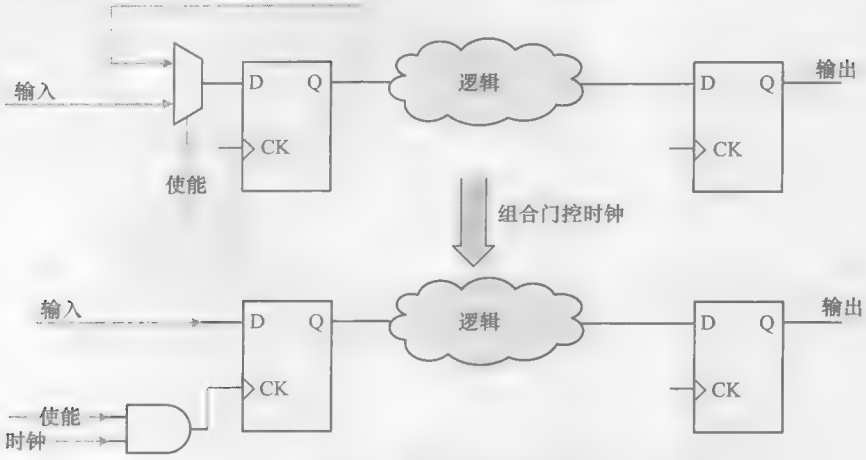


图 5.5 组合门控时钟

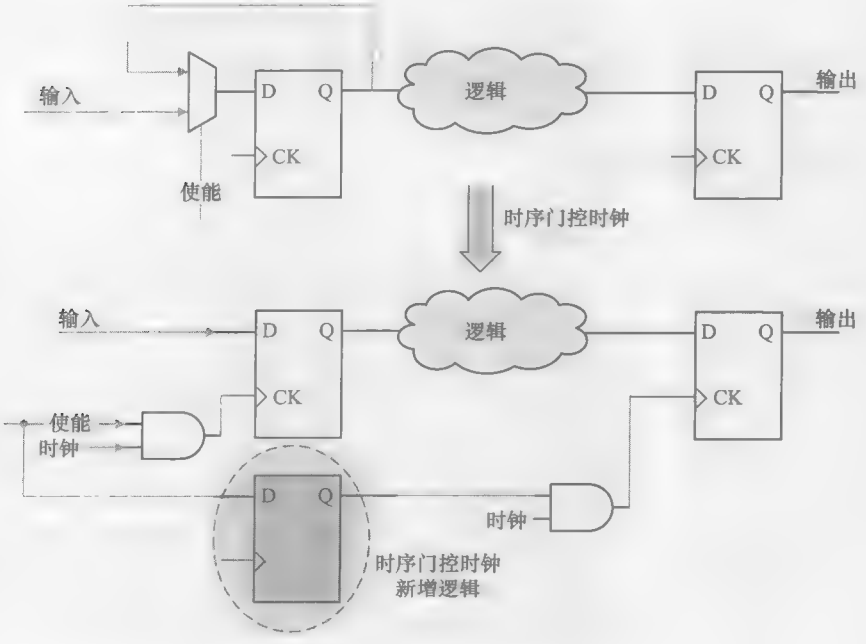


图 5.6 时序门控时钟

使用时序门控时钟最大的挑战在于识别出流水线上“多余的”或“不关心”的状态，一旦这一工作完成，使用时序门控时钟就可以降低大量功耗，典型情况下可以降低 15% ~25% 的开关次数[14]。

按照文献［14］，时序门控时钟转换只在有多于16个触发器的情况下才更有效率。

5.5.2 动态电压频率调节

动态电压频率调节（Dynamic Voltage/Frequency Scaling，DVFS）是提高系统能效的流行方法。在对频率不敏感的应用阶段中降低时钟速率和供电电压，可以在性能适度损失的情况下大幅降低功耗。

5.2节提到，功耗正比于供电电压的平方。因此减少供电电压将会使功耗相应降低四分之一。但是降低供电电压也会降低系统的性能，所以需要作出权衡。

对于像移动手机、PDA这样的手持设备，在设计目标上存在着固有的冲突，即它们应当能使电池使用时间尽可能长，但是作为智能设备又需要性能强大的处理器，这将比简单设备消耗更多的能量，从而减少电池使用时间。尽管随着半导体技术和电池技术的发展，使得微处理器在单位能耗下能提供更强大的运算能力，但是在性能和电池寿命方面进行基本的权衡仍然非常关键。

对于典型应用，一般只在很小的时间比例中需要高性能，而在其余的大多数时间里使用低性能、低功耗的处理器就足够了。在处理器不需要全速运行时，可以简单地通过降低其频率来达到这个目的。

当今使用CMOS工艺的绝大多数处理器的最大频率往往与供电电压相关，所以在低频时处理器可以工作在低供电电压下。因此，DVFS可以通过频率和电压调节节省大量能量。

在实时嵌入式系统中，往往不能直接使用目前已知的DVFS算法，因为改变处理器的操作电压将会影响到任务的执行时间并违反时效性要求。

有些DVFS能合并到OS调度程序和实时嵌入式系统任务管理服务中，这样既能通过DVFS节省功耗，又能保证对时效性的要求。

特别是，动态电压调节（Dynamic Voltage Scaling，DVS）依赖特殊的硬件，而使用一个可编程的DC-DC开关电压调节器，一个可编程的时钟产生器和一个宽操作域的高性能处理器提供了两全其美的方式。在需要峰值计算载荷时，处理器在正常电压和频率下操作。当负载降低，在能满足对运算能力的要求的条件下频率也降低。

随着技术的进步，现在设计者可以将越来越多的核集成到单芯片上。微处理器设计现在已经从对全芯片电压/频率控制转变为更细粒度的方式，

从而增加了能效。例如，AMD 的四核皓龙处理器允许对所有四核进行独立频率控制，共享的 L3 缓存和片上北桥，DDR 接口和四个超传输链接[15]。

总之，通过根据运算负载动态调整处理器/系统的电压和频率，DVS 能提供峰值满足运算要求的性能，通常这种方式降低功耗（包括每单元计算的能量）所带来的好处能体现在低性能处理器上。

5.5.3 基于缓存的系统体系结构

对大多数 DSP 应用，快速傅里叶变换（Fast Fourier Transform，FFT）算法要求频繁访问系统存储器的数据，而这样的存储器能效并不高。增强型 FFT 体系结构需要在系统存储器或 RAM 和处理器之间增加缓存。

这种方案的最初目的是在处理器需要之前，预先将相关数据从主存中取到缓存中。使用小范围的缓存能使计算能耗大量下降，极大地提高了 FFT 的能效。

5.5.4 对数 FFT 体系结构

对于要大规模运算的应用，使用对数系统（Logarithmic Number System，LNS）比使用线性系统更好。LNS 在降低平均位元活跃度的同时用加法和减法实现乘除运算，使其效率比线性系统更高。因此基于 LNS 实现 FFT 可能节省大量功耗。负面影响是加法器和减法器的宽度会增加——导致需要以指数级增大的查找表。

5.5.5 异步（无时钟）设计

对于基于时钟体系结构的同步时钟设计（见图 5.7）来说，时钟分布消耗掉了大部分的能量。传统的设计方法学会形成大规模时钟树结构，这样便潜在地增加了 SoC 的平均功耗。时钟分配网络的设计也耗费了设计者的大量精力。最可能存在的问题的是时钟偏移，即时到达电路不同部分的时间的差异。当电路规模较大并较慢时，时钟偏移无关紧要。但是随着电路工艺缩小和速度增加，时钟偏移的差异就会变得更加明显，需要额外的设计时间和电路来解决这个问题。将时钟分布到整个芯片上并不是一件容易的事，而且可能会产生不规则的版图。此外存在的另一个问题就是相当大的功耗。

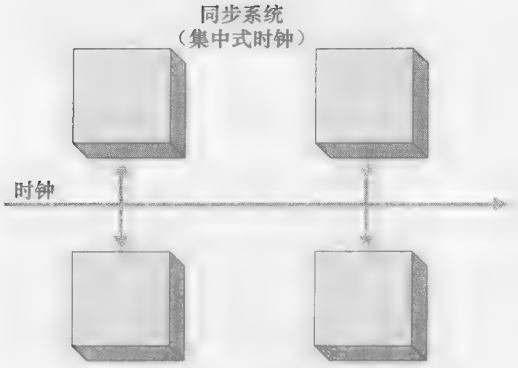


图 5.7 集中式时钟的同步系统

由于时钟会导致以上一系列问题，将其从设计中移除是一件很有诱惑力的想法。这就是异步设计的基本意图。然而，并不能简单地直接移除时钟，因为仍需要对电路操作进行某种控制。异步电路本质上进行自我控制，因此也称为自定时电路。

图 5.8 中是一个异步系统，其中两个模块使用握手接口进行交互。

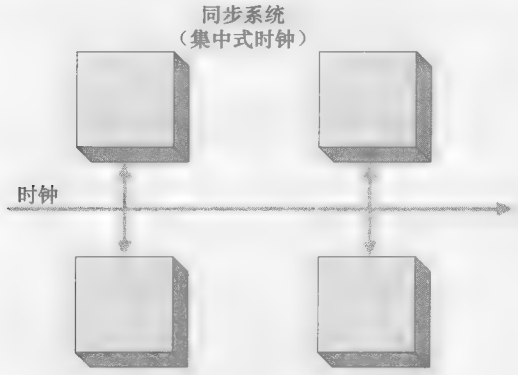


图 5.8 基于握手接口的异步系统（无全局时钟）

移除时钟使得能效提升。时钟耗费了许多能量，特别在大规模高速系统中，所以将其移除使得能效大大提高。除此之外，因为未工作元件几乎不消耗能量，所以异步电路的动态功耗接近于 0。

异步电路基于依赖延迟不敏感编码（delay in-sensitive encoding）的信号交换接口，其中最流行的是双轨道编码。

双轨道编码用两根线传输每个数据位，因此称为双轨。（单轨只使用一根线传输每个数据位。）

在双轨道编码中，一根线用于表示逻辑1，另一根表示逻辑0。两部分能可靠地彼此通信而不受线上延迟的影响。这种协议对延迟是不敏感的。编码方式如下：

“LL”=“无数据”，“LH”=“0”，“HL”=“1”这里L=逻辑“0”且H=逻辑“1”

这里0或1表示有效数据。“11”表示无效。

注意，这里 n 位数据通信需要 $2n$ 条线，每位都是自定时的。

本节所描述的编码方式是四相双轨道编码。双相双轨道编码也使用两根线表示1位数据，但是信息按事件进行编程（0到1，1到0）。在一根线进行传输时，接收新的码字 不存在空值（即没有“00”）。一条有效消息应答后紧随另一条有效消息。

也有其他对延迟不敏感的编码和基于事件的信号，应基于引脚和能效进行选择。

按照《国际半导体技术路线图》（ITRS，ex. SIA），1999版：

> 随着时钟速度很可能超过5GHz，并且片间通信开支占用5～20个时钟周期，需要一种建立层次化时钟速度的方法，实现本地同步和全局异步连接。这需要既能处理异步、多周期互连又能处理本地同步、邻近高性能通信的工具。

异步电路的进一步细节已经超出了本书的范围。

5.5.6 电源门控

与电压门控类似，电源门控可以在模块不使用时暂时将其关闭。

对于使用90nm以下工艺设计的器件，晶体管漏电流以指数程度增长，这使得设计满足预期的功耗目标成为一个挑战。减少过度漏电以保证电池寿命很重要，尤其对于使用电池的手持设备。电源门控是解决这个复杂挑战最有效的技术，它可在逻辑模块不操作时将其关闭。

电源门控（或电源切断技术）通常指在芯片上加入开关以根据应用要求选择性切断供电电流。设计者可使用两类电源门控：细粒度电源门控和粗粒度电源门控。

5.5.6.1 细粒度电源门控

在细粒度电源门控中，在每个门和地之间存在一个开关晶体管。这种

方式允许在不使用某些功能时关闭与地的连接。可以对库中的每个元件进行这种控制。

添加开关晶体管所引起的主要负担增加在库 IP 提供者或标准单元设计者身上[16]。通常这些元件的设计遵循普通标准单元规则并能被 EDA 工具用于实现。

电源门控的尺寸必须能满足任何情况下对开关电流的需求。这种门必须较大以便不会引起可测量的压降（IR）。此外对头部开关（P-MOS）和脚部开关（N-MOS）也要作出选择，如图 5.9 所示。通常在所提供的开关电流相同时，脚部开关的面积更小。动态功耗分析工具能精确地测量开关电流，并能预测电源门控的尺寸。

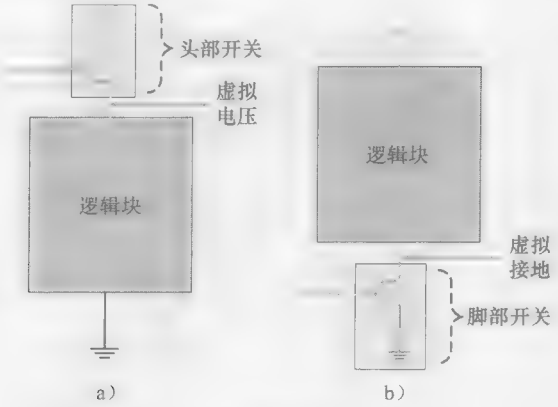


图 5.9 头部和脚部电源门控

使用细粒度电源门控是一门优雅的艺术，可以将漏电减少为原来的 $1/10^{16}$。在仅仅使用多电压优化方法不能满足功耗要求的情况下，这种降低功耗的方法是一种很有吸引力的技术。

5.5.6.2 粗粒度电源门控

在粗粒度电源门控中，电源门控晶体管是供电网络的一部分而不是标准单元的一部分。粗粒度本质上创建一个电源开关网络，各组开关晶体管能并行地将整个模块打开或关闭，如图 5.10 所示。

与细粒度电源门控不同，粗粒度方式不完全依赖于库的质量，而 EDA 工具对其处理能力影响更大。

粗粒度电源门控的操作与细粒度电源门控的操作相同。然而，对该机制的实现和分析是很复杂的。置于电压关闭区域的备用晶体管的大小和数

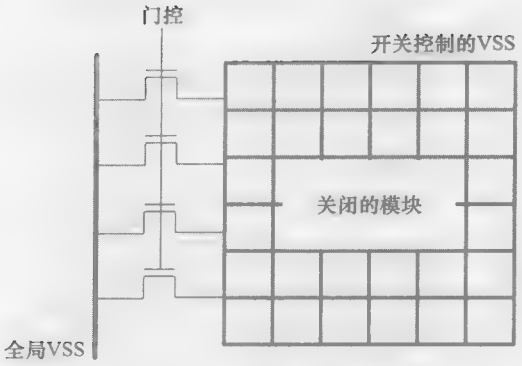


图 5.10 粗粒度电源开关

目会影响对该区域的驱动能力。这会将会导致 IR（电压）降幅变化并使性能衰弱。当所有的头部开关同时再次打开，从 VDD 到 VSS 就会有瞬间充电/放电电流和短路电流产生。该行为如图 5.11 所示。

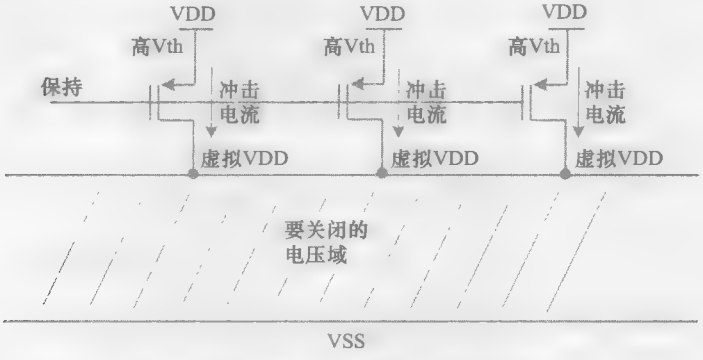


图 5.11 粗粒度电源门控的冲击电流

这种电流总和称为上电冲击电流。对于避免器件故障和潜在的芯片失效，这种电流是个关键。这也是粗粒度电源门控实现主要考虑的方面。

注意，不管实现哪种类型的电源门控，都需要改变 RTL，以便设计一个电源控制器对需要关闭的模块进行控制，而且该模块的供电电压需要加以考虑。百万门规模的 ASIC 很轻易就有 20 个以上的电源域。在这样大规模的设计中很难控制使用单一的细粒度或粗粒度技术，因此实际上可能会混合使用这两种技术。

为了最小化漏电流，门控电源晶体管常常使用高阈值电压的单元。粗粒度电源门控通过优化低开关率的单元进一步增加了灵活性。这些单元也

可以用填充元件取代来保证电源分配网络的完整性。

粗粒度电源门控技术能极大减少功耗。在本书出版时，只有高级 EDA 工具能支持这种方法。在门单元大小的选择、布局布线、同步开关分析和门控信号转换速率方面的比较涉及复杂的优化过程。工具应当能识别多电源域设计要求，以便能在保证硅质量前提下进行优化。

电源门控也需要在“关闭”域和“开启”域之间的逻辑或信号上加入隔离单元以保持设计完整性并避免功耗损耗。当一个模块断电后，它输出给另一个仍然带电模块的信号就可能变为浮动状态，浮动值可能在阈值电压内并产生不期望的电流。带电模块输入处的隔离单元将下电模块输出的信号固定在1或0值。通常可用简单的或/与逻辑作为输出隔离器件。隔离单元要么在 RTL 中插入，要么在指定参数和要插入的模块后由 EDA 工具自动插入。现在的工具有足够的能力可以根据要求将这些单元插入合适的层次。一部分在综合时插入，其余的在布局布线时插入。

5.5.7 多阈值电压

多单元库有助于处理漏电和动态功耗的问题。典型的多单元库包含至少两组功能相同但阈值电压不同的单元。高阈值电压单元较慢但是漏电较少；相反，低阈值电压单元更快但是漏电较多。

高阈值电压单元通常会比低阈值电压单元降低50%的漏电，并且没有任何如面积增加这样的副作用。

另一方面使用这种技术会增加生产的复杂性，也会增加设计时间。对设计不恰当的优化可能导致使用更低的 Vt 单元，因此使功耗增加。

对于比较使用各种不同的 Vt 单元能否达到所期望的性能这一过程，最好在综合和布局优化阶段进行。逻辑综合或优化过程的门级映射阶段由综合工具实现，布局优化由物理实现工具负责处理。设计者可能根据设计目标使用不同的策略。

如果最终的设计目标是满足性能，设计者可以使用低阈值电压库进行第一遍综合以达到最好的性能来满足对时序的要求。然后对于设计中不需要最高性能或低阈值电压的单元用高阈值电压单元进行替换，以减少整体功耗和设计的漏电流。这是多阈值设计技术最常见使用的方法，因为大多数应用是以满足时序要求为前提条件的。在综合时使用低阈值电压库速度更快，并且能得到更小的设计面积。综合工具在使用大量高阈值电压单元时可能会耗费更多的运行时间并产生较大面积的设计。

然而，如果设计的主要目标是功耗，而面积是其次的，在进行第一遍综合时使用高阈值电压单元更合适，然后找出关键路径并用低阈值电压单元进行替换，以达到最终的性能要求。

5.5.8 多电压供电

从5.2节中的动态功耗方程中可以看出功耗与供电电压的平方成正比。在多电压供电（MSV）设计中，设计可以分割为独立的“电压岛”或“电压域”，根据每个区域对时序的要求而使用不同的供电电压。

一种划分方式是将对时序要求严格的模块置于标准电压下操作，对于90nm工艺是1.0V。对于时序要求不那么严格的路径可以安排到其他区域，供电电压也可以下降至0.8V，这部分设计的动态功耗可下降36%。

在过去，这种方式为物理设计阶段增加了额外的复杂度。设计者通常需要手动插入指定的转换单元，称为电平移位单元，使信号可以在不同电压域之间传输。对于数百万门的ASIC，可能会有上千个信号跨电压域传输，这就增加了设计风险的发生。现在很多EDA公司都能提供可以自动插入电平移位单元的工具，使这种方法越来越可行。

5.5.9 存储器电源门控

在典型的SoC中，SRAM消耗了总功耗的1/3，其余部分由时钟树和随机逻辑消耗掉。所以对于好的电源管理策略，存储器架构是个关键因素。

最简单的形式是在不使用存储器矩阵时将其关闭。需要在功耗方面对使用单个大规模存储器矩阵或使用多个小存储器进行比较。

如果SoC只需要一小部分存储器常开，而其余部分存储器可以根据操作模式开关，就可以将存储器分割成多个部分。小部分存储器处于电压“开”状态，其余大部分存储器只在执行密集型运算时打开。

此外注意，如果将大块存储体划分为多个小块后，读周期总数仍是相同的，但是每个读周期所消耗的功耗会大大降低。

另外一种技术称为基底偏压存储器（body-biasing memory）。在这种方法中，设计者在不使用存储体时将其反向偏置，本质上便提高了阈值电压并降低了漏电功耗。

另一种流行的方法是使用多模式给存储器供电。在该技术中，设计者利用存储器和几种供电模式。许多设计利用双功能存储器，以便在处理器对存储器进行读写操作时，存储体使用全电压供电以保证操作得以进行。然而，在存储器不需要读、写时，处理器可以编程降低其供电电压，使之仅能保持数据不丢失即可。

还有一种在封装级降低存储器功耗的方法是使用堆叠式存储器，即将存储器堆叠在裸片上面。堆叠式存储器显著地降低了互连电容，并将存储器功耗降低近30%。至少，对于性能关键型应用，需要较大的存储器带宽（如图像、多媒体和调制解调等），可以使用将相关存储器堆叠式封装的方式，而操作系统或其他应用则可以置于外部存储器中。

5.6 在寄存器传输级降低功耗

在大规模ASIC中，在RTL（寄存器传输级）完成时至少80%的功耗已经确定。后端流程不能解决所有功耗问题。需要系统性地直接从RTL中或从映射结果中寻找降低功耗的机会。对于有缺陷的设计后端流程是无法修复的，如果系统架构出了问题，关键路径就无法达到闭合。后端流程无法修复微架构，微架构和RTL代码风格对于动态与静态功耗有极大的影响。

因此，有效的方法学要求在综合前的RTL阶段中就将与功耗相关的所有问题解决。

5.6.1 状态机编码和解码

在各种状态机编码类型中，格雷码是最适合低功耗设计的。

图5.12将二进制编码状态机与格雷码状态机进行比较。对于二进制编码，在状态转换过程中可能有多个触发器发生翻转，例如，从状态D（“011”）到状态E（“100”），这比在状态转换过程中每次只有一个触发器变化的格雷码要消耗更多的能量。

此外以格雷码编码的状态机也消除了依赖于状态的组合等式中存在毛刺的风险。

出于某些原因，如果对状态机使用了差异性较大的编码风格，仍然可以通过降低翻转为较多状态的切换频率，在状态转换时降低功耗。例如，

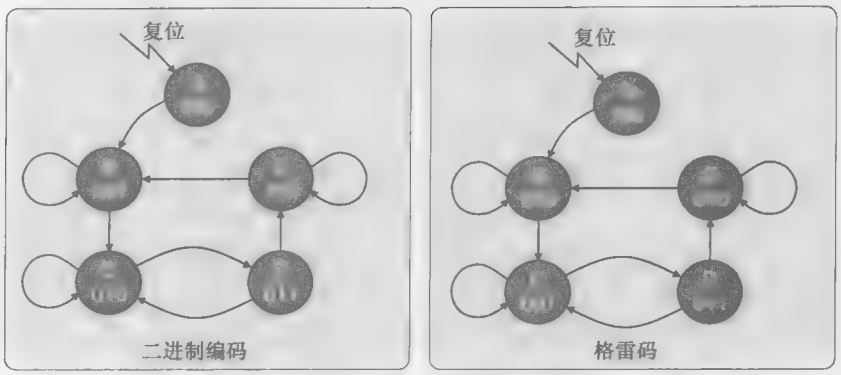


图 5.12 低功耗设计中二进制编码与格雷码的比较

对于16位状态机，有256个状态值，其中一般只有一部分会使用到，这部分中一些状态会比另一些状态在电路操作过程中更可能出现。所以，如果状态机中有30%的转换发生在从状态“0101”变为“1010”，有4位发生变化，这就导致4个状态寄存器及其相关组合逻辑发生了转变；对于从状态“1010”变为“0100”，只有一个状态位发生了变化。这样可将寄存器功耗降低10%，而组合逻辑部分可能会降得更多。

另一种是如文献［27］中的作法，将有限状态机分解以达到低功耗效果。基本设想是将有限状态机（FSM）的状态转换图（STG）分解为两个，它们共同作用以达到与原来状态机相同的效果。这样做之所以能降低功耗，是因为如果两个子FSM之间没有转换发生，那么只有一个子FSM需要供给时钟。

这项技术遵循标准分解结构。根据各状态之间的高概率转换，和与其他状态之间的低概率转换将状态划分为各子集。状态子集组成在大多数情况下常开的子FSM。当小的子FSM被激活时，可以关闭其他较大的子FSM。

最终，因为绝大多数时间内只需给较小且更有效率的子FSM供给时钟，所以降低了功耗。

5.6.2 二进制数表示法

在大多数应用中，用补码来表示二进制数往往比有符号数更常使用。然而，对于某些特殊应用，在切换过程中有符号数更有优势。

图5.13中是对“0”和“1”分别用补码和有符号数表示的方法。

补码	有符号数
"0" => 00000000 "-1" => 11111111	"0" => 00000000 "-1" => 10000001

图 5.13 二进制数的补码和有符号数表示法

对于某些只使用积分器进行求和的应用，补码表达法在 0 到 1 的转变发生时所有位都会发生变化（有较高的开关功耗），与之相比有符号数只有两个位发生了变化。

5.6.3 门控时钟基础

门控时钟在 2.5 节中已经有所介绍，这里从 RTL 编码的角度对门控时钟问题加以描述。

让我们考虑一个 32 位寄存器 "test_ff"，在载入使能 "load_cond" 为真时会写入 32 位输入数据值，否则该寄存器的值保持不变。图 5.14 是该逻辑的 RTL 代码。图 5.15 为相应的电路实现。

块A:

```
always @(posedge clock or negedge reset_b)
  if (!reset_b)
    test_ff <= 32'b0;
  else
    test_ff <= test_nxt;
```

块B:

```
assign test_nxt = load_cond ? test_data : test_ff;
```

图 5.14 测试逻辑的 RTL 代码

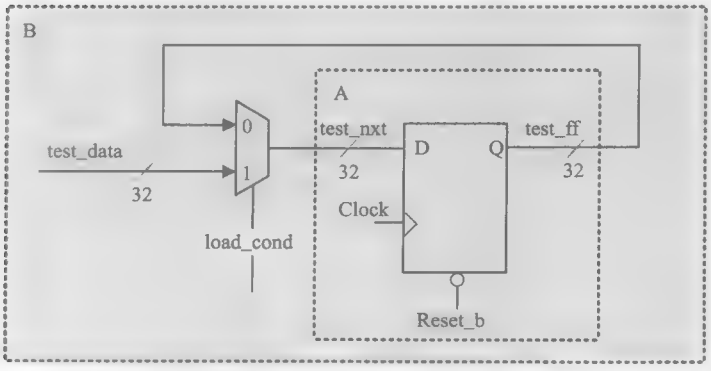


图 5.15 测试逻辑的逻辑实现（反面例子）

注意，根据RTL的写法，在clock线上不能推出任何门控时钟。某些后端工具可能会在该层次之上或者在预扁平化之后产生门控，但是我们不应依赖于这种方法。

图5.16中是以另一种风格编码的相同逻辑的范例，以便可以自动推出门控时钟。

块B：

```
always @(posedge clock or negedge reset_b)
  if (!reset_b)
    test_ff <= 32'b0;
  else if (load_cond)
    test_ff <= test_data;
```

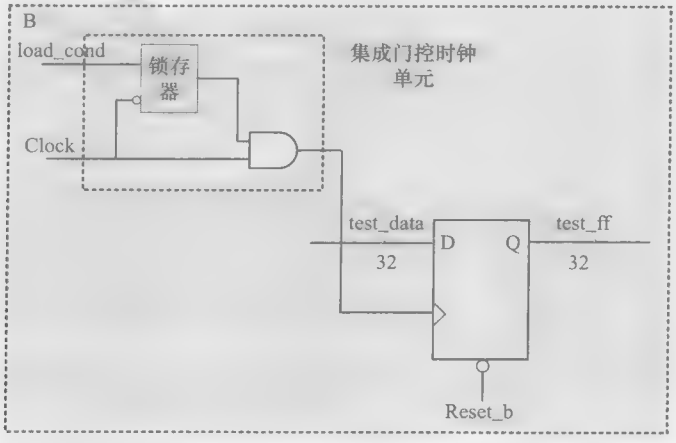


图5.16 测试逻辑的逻辑实现（范例）

用RTL这种编码方式，HDL编译器可以看到模块的整个图景并能识别“load_cond”为32个寄存器位所共享。此外后端环境设置时也会以加入一个门控时钟（集成化的库单元）来取代32个多路复用器门。集成时钟单元在扫描模式下通常会旁路掉（在图5.16中未显示）。

对某些综合过程未能识别的情况，可以明确地指定门控时钟对所有功能进行动态控制。

与门控时钟类似，门控信号（在2.5.3节中描述）和数据通路重组（在2.5.4节中描述）等都应当在编写RTL时予以考虑以便达到进一步降低功耗的目的。

5.6.4 独热码多路器

在 RTL 中有许多推出多路器的方法。“case”语句、“if”语句和状态机一般都能实现这种效果。表示多路器（MUX）最常用的方法是使用二进制编码，如图 5.17 所示。

```
case (SEL)
   2'b00: OUT = a;
   2'b01: OUT = b;
   2'b10: OUT = c;
   2'b11: OUT = d;
endcase
```

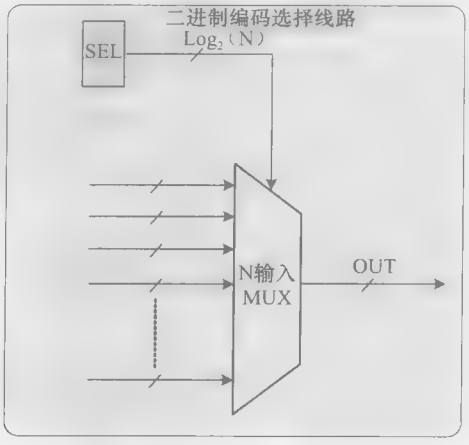


图 5.17 MUX 选择线路的二进制编码

注意，如果 MUX 的每个输入是多位总线，就会产生明显的开关过程，由此产生功耗。

如果对“case”条件进行编码时按图 5.18 中所示的独热编码方式，而不是二进制编码方式，输出就会更快、更稳定，而且在初期就能将未选中

```
case (SEL)
  4'b0001: OUT = a;
  4'b0010: OUT = b;
  4'b0100: OUT = c;
  4'b1000: OUT = d;
  default: OUT = "X";
endcase
```

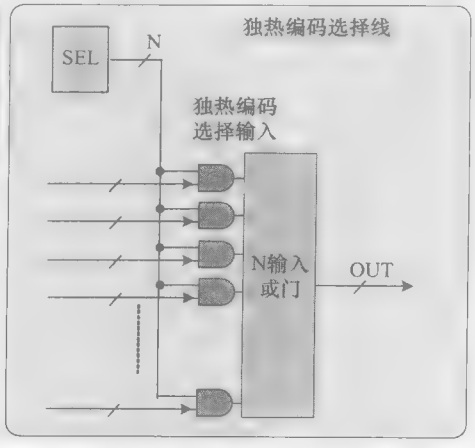


图 5.18 MUX 选择线路的独热编码

总线掩藏掉，因此实现了低功耗效果。

一般数字设计中都会存在多路器，因此避免或掩蔽掉伪转换的发生能有效地降低功耗。

5.6.5 除掉多余的转换

在没有设置默认状态的情况下，总线数据常常会发生没有意义的转换。如果转换后的数据未被真正采样，那么它就是多余的，将这样的转换去掉显著可以降低功耗。

图5.19中是含有多余转换的例子，这个例子在读入所有输入信号时消耗了能量，但是最终的输出却未使用。

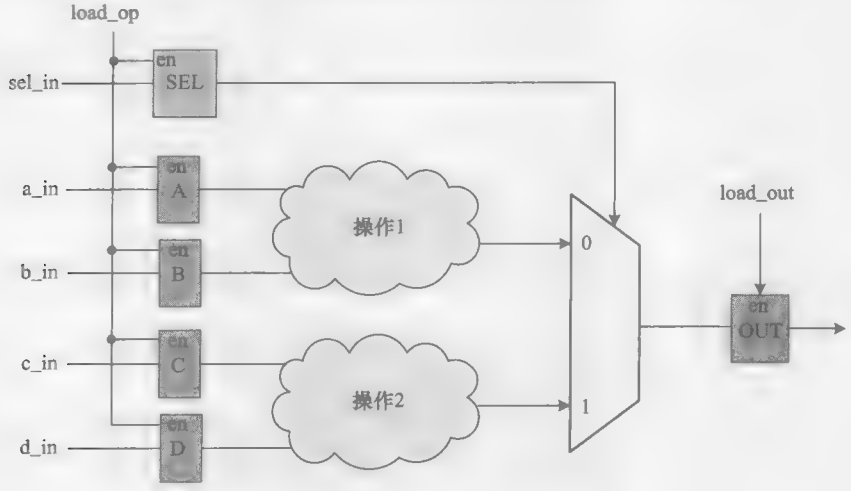


图5.19 冗余转换消耗了能量

注意，如果“load_out”不会置为有效，那么与之相关的“load_op”也不应置为有效，这样可以节省一部分功耗。

图5.20是将多余的转换进行压缩的方案。仅当“SEL”为0时才读入“A”和“B”，而仅当“SEL”为1时才读入“C”和“D”。

图5.21中同样的同一数据输入到所有目标模块中，但是只有“data_sel”有效的模块才接收该数据。由于规定“data_sel”每次只有一位有效，而所有四条总线分支上都产生了数据切换，这样就导致了不必要的功耗。

图5.22是修改后的逻辑，消除了多余转换。新增了一些逻辑门，用来取消未选择的总线分支的转换，这些逻辑门会增加一些功耗，但保证了只有最终选中的目标才有数据切换发生，这样就降低了功耗。

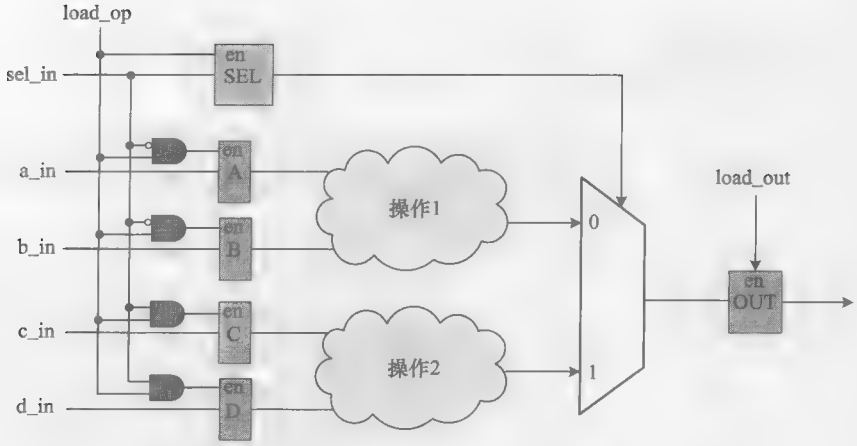


图 5.20 压缩冗余转换以节省能量

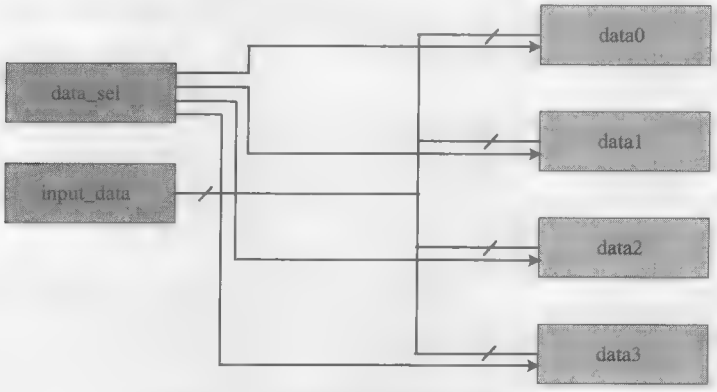


图 5.21 点对多点式总线的冗余转换

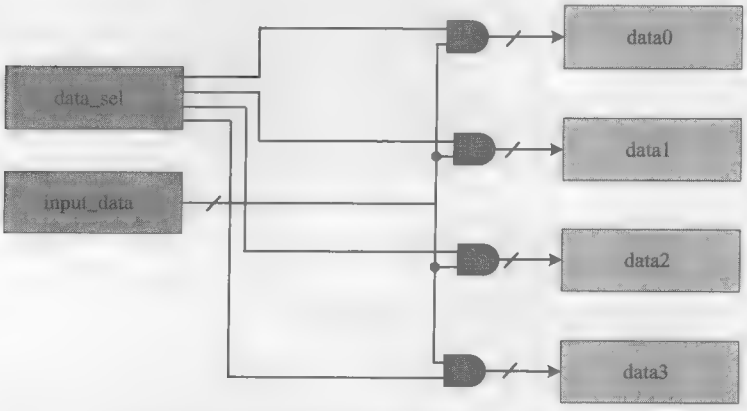


图 5.22 在点对多点式总线中压缩冗余转换

5.6.6 资源共享

对于涉及较多算术运算的设计，如果有同样的操作在多处使用，必须避免相应的运算逻辑在多个位置重复出现。图 5.23 是没有共享资源的例子。

```
always@(*)
 case (SEL)
  3'b000: OUT = 1'b0;
  3'b001: OUT = 1'b1;
  3'b010: OUT = (value1 == value2);
  3'b011: OUT = (value1 != value2);
  3'b100: OUT = (value1 >= value2);
  3'b101: OUT = (value1 <= value2);
  3'b110: OUT = (value1 < value2);
  3'b111: OUT = (value1 > value2);
 endcase
```

图 5.23 在逻辑中未使用资源共享

使用重复逻辑会增大面积，而且增加功耗。

图 5.24 显示了在各分支条件中只使用一个比较器“==”和算术比较器“>”就实现了上面同样逻辑的功能。

```
assign cmp_equal = (value1 == value2);
assign cmp_greater = (value1 == value2);

always@(*)
 case (SEL)
   3'b000: OUT = 1'b0;
   3'b001: OUT = 1'b1;
   3'b010: OUT = cmp_equal;                    // ==
   3'b011: OUT = !cmp_equal;                   // !=
   3'b100: OUT = (cmp_equal || cmp_greater;    // >=
   3'b101: OUT = !cmp_greater;                 // <=
   3'b110: OUT = !cmp_equal && !cmp_greater;   // <
   3'b111: OUT = cmp_greater;                  // >
endcase
```

图 5.24 在逻辑中使用资源共享

5.6.7 使用行波计数器来降低功耗

2.4.3 节已经讨论过行波计数器的缺点和限制，但在低功耗设计中使用它会取得很好的效果。本节将讨论使用行波计数器所面对的挑战以及可

能的解决方法，以使其更有效地应用在低功耗设计中。

再次参考2.2.1节中的图组成四位等效计数器，时钟偏移的细节如图5.25所示。

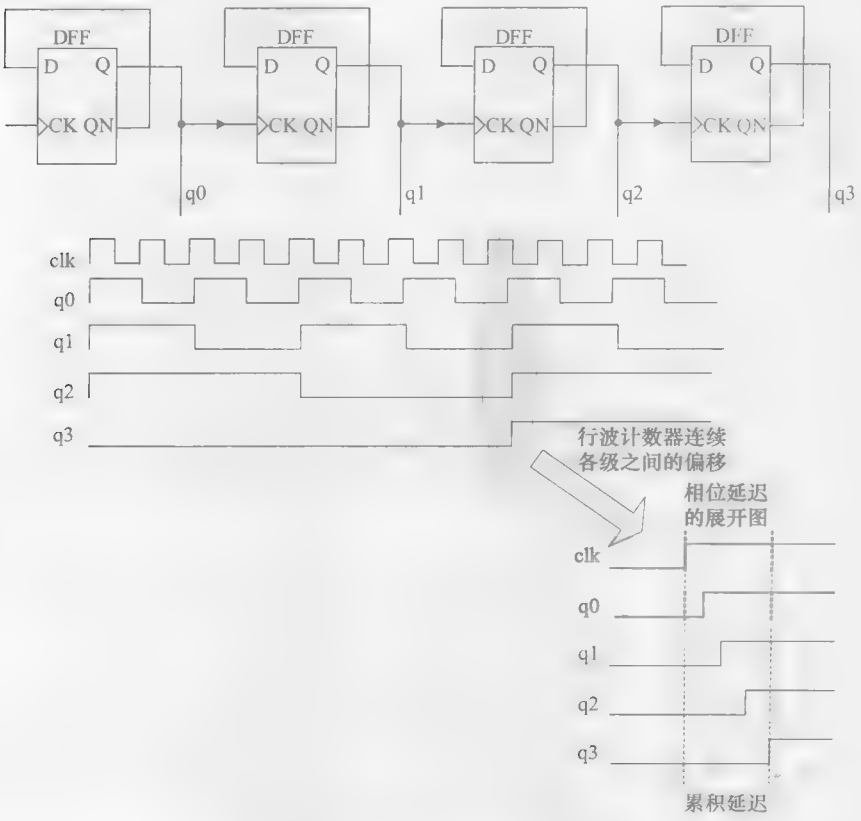


图5.25 四级行波计数器中的相位延迟

每个阶段都进行二分频操作。因为时钟像波浪一样行进穿过系统中各寄存器，所以称为“行波计数器”。由于每个寄存器都存在传播延迟，因此较高有效位相对于较低有效位更晚发生变化。

在时钟行进过程中计数器可能产生不正确的值（由于毛刺）在最高有效位变化时，会产生最坏的影响。

在图5.26中是计数器从“0111”变为“1000”的时序图，在这个过程中4个位都发生了变化。在时钟传递过程中会出现短暂的伪输出值。计数器电路会快速在“0111”、“0110”、“0100”、“0000”、“1000”之间变化，即值为7、6、4、0，然后到8，而不是干净地从“0111”输出到“1000”。

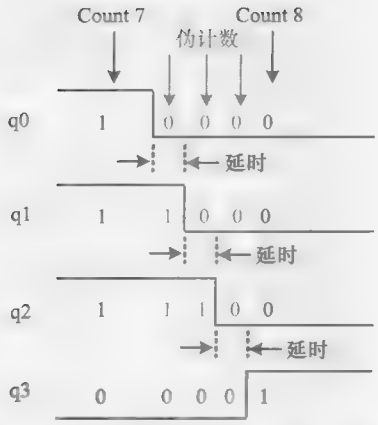


图 5.26 四级行波计数器中的相位延迟

在许多使用环境中，这种效应是在可接受范围内的，因为行波是快速发生的。如果用计数器的输出驱动发光二极管（LED），根本不会有任何影响。但如果将该信号用于驱动多路选择器，微处理器（计算机）电路中存储器的索引指针，或者任何其他伪误差可能导致错误输出的情况，便是无法接受的。

行波计数器对于静态时序分析是一个巨大的挑战，因为行波计数器中每个阶段都产生了一个新的时钟域。这就需要静态时序分析工具处理更多的时钟域，从而会消耗更多的时间。

同理，行波计数器也会为扫描链的插入增加难度。可以在扫描模式下使用多路器对这些寄存器的时钟进行选择，从而使其在扫描模式下处于同一个时钟域中。但是这样会对缺陷覆盖率产生影响，因为新引起的时钟多路器形成了一条无法扫描到的路径。

注意

基于上面所提到的种种挑战，数字设计者应该在有限的情况和严密的控制下使用这种技术。

有一些方式可以使行波计数器更可靠，可用性更强。图 5.27 中的电路可以使计数器不再产生毛刺。

接收电路使用低有效使能输入信号，只在时钟为“低”时才读入 4 位计数器的值。一旦时钟脉冲置“高”，接收电路就停止响应计数器电路的输出。因为计数电路是正沿触发的，所有计数行为都发生在时钟从低到高的过程中，这样就使接收电路在计数器的 4 位输出信号切换稳定之前一直处于无效状态。图 5.28 描述了这种行为。

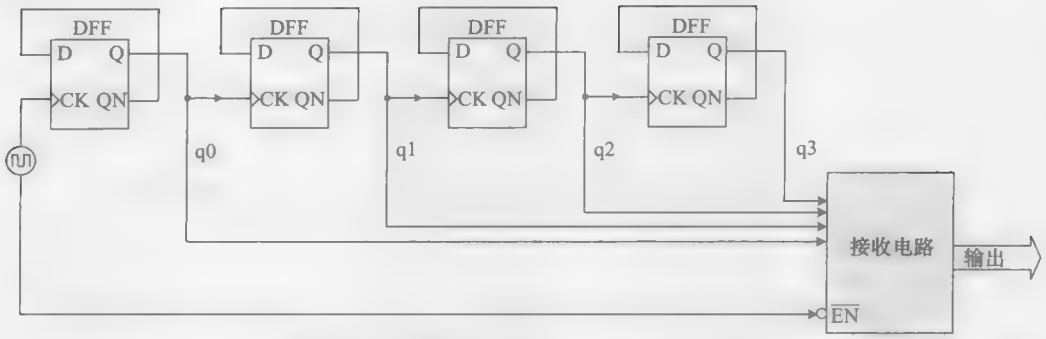


图 5.27 行波计数器输出不含毛刺的转换

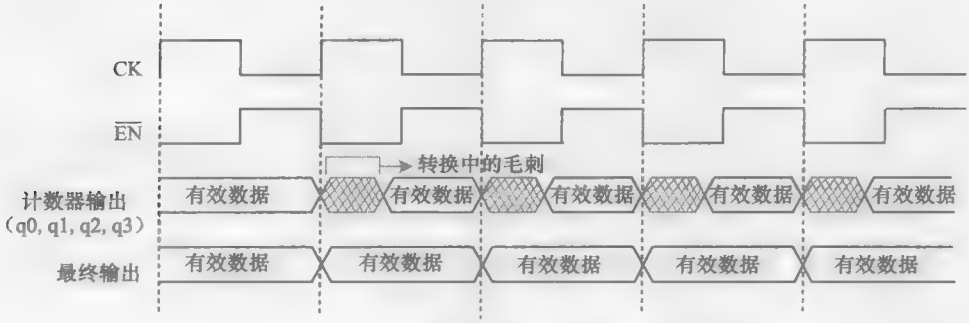


图 5.28 不含毛刺的行波计数器的时序图

直到时钟信号返回低状态，接收入电路才会开启，这样可以保证所有行波都处于安全稳定状态时新计数值才会读入接收电路。这里时钟信号的“高”时间是一个关键参数：它至少要与计数器的最大行波周期同样长。否则，时钟信号会过早开启接收电路，而这时行波过程仍在进行中。

总之，行波计数器能减少电路的漏电流并降低电路的功耗，但是在使用时必须非常小心。虽然行波计数器实现起来看着很简单，但是会对可测性和缺陷覆盖率产生影响，所以必须要在使用这种异步计数器方法前从正反两方面加以权衡（如在本节中所提到的）。

5.6.8 总线反转

在当前数据和下一个数据之间的汉明距离大于 N/2（N 是总线宽度）时，就将下一个数据反向后传输，以减少总线上出现转换的位数量，这就是总线反转编码。这种技术对于减少大容量总线上的转换次数很有效。

该总技术在传输数据的同时也需要传输额外的一个控制位，以表示所传输的数据是否是已反转的（如图5.29所示）。

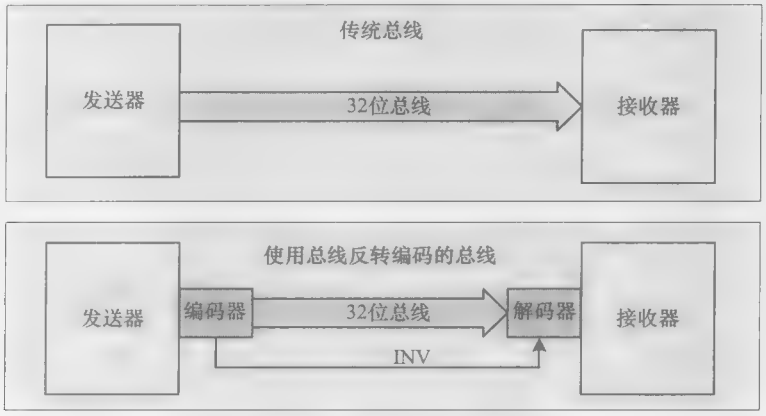


图5.29 总线反转编码

如图5.30中的例子所示，在总线反转后转换总量的区别是很明显的。

图5.30 总线反转示例

5.6.9 高活跃度网络

处理这类设计的思路是将活跃度较高的网络和活跃度较低的网络区分开来，然后将它们置于逻辑云中尽可能深的位置。

图5.31所示的逻辑云是 $X_1 \cdots X_n$ 和 Y 的函数。$X_1 \cdots X_n$ 的变化频率较低，而 Y 是高活跃度网络。在右图的具体实现中，把逻辑云复制了一份，

一份中假设 $Y=0$，另一份中假设 $Y=1$，然后根据 Y 的值决定选择使用哪一个。由于针对一个固定的 Y 值，通常这两份新逻辑云的规模会有所减小。

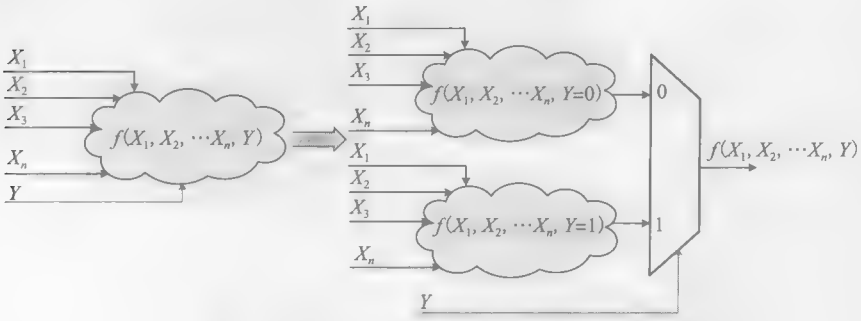


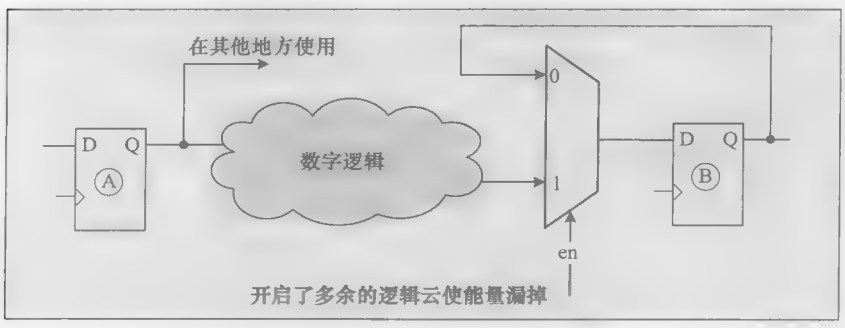
图 5.31 将高活跃度网络与静态网络分开

某些降低功耗的工具能给设计加上驱动输入向量，然后对活跃的网络进行分析，并自动进行优化。

5.6.10 启用和禁用逻辑云

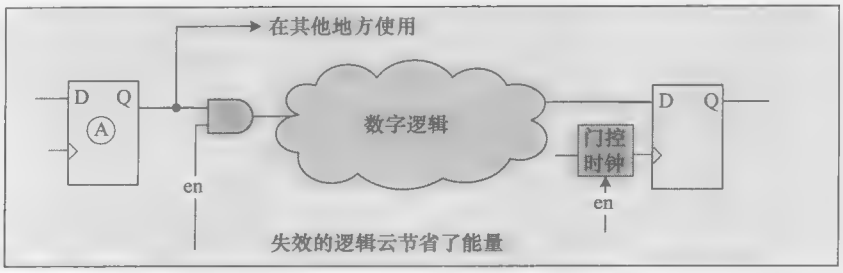
在操作大规模逻辑云（包括宽加法器、乘法器等）时，往往会在需要时才将其打开。

在图 5.32a 的实现方法中，只有触发器“B”得到使能信号。触发器“A”并没有关闭，因为设计中的其他地方用到了其输出，所以使得整个逻辑云启用，并且浪费了能量。图 5.32b 是另一种实现方法，将使能信号移至逻辑云前，并在不需要使用逻辑云时禁用它。



a)

图 5.32 使逻辑云失效以降低功耗



b)

图 5.32 （续）

5.7 寄存器级低功耗技术

每代新的硅工艺都会相应带来功耗的降低。以因子 Φ 缩减的工艺节点会将功耗降低至 $1/\Phi^3$。工艺在迅速缩小至深亚微米级，不久就会到达40nm。随着各种新 CMOS 生产工艺的出现会将功耗降低到一个新的水平。

5.7.1 技术水平

在本章中，前面提到的所有降低功耗的技术都可以由电路设计人员直接实现。通过更先进的硅处理技术，可以额外降低功耗。

5.7.2 版图优化

在版图阶段进行优化可以显著降低功耗。理想的优化意味着所有有直接关联的模块在硅片上都应紧密挨在一起。长布线会增加功耗。遗憾的是，复杂的 SoC 使得越来越难以达到这种结果。

5.7.3 衬底偏压

由于漏电流是阈值电压 V_{th} 的函数，如图 5.33 所示，衬底偏压也称作“反偏压”能减少漏电功耗。采用这种技术，可以将衬底或合适的阱区电压偏置，以提高晶体管阈值，因此减少了漏电。在 PMOS 中，指将晶体管衬底偏置到高于 V_{dd} 的水平。在 NMOS 中，指将衬底电压偏置到低于 V_{ss} 的水平。

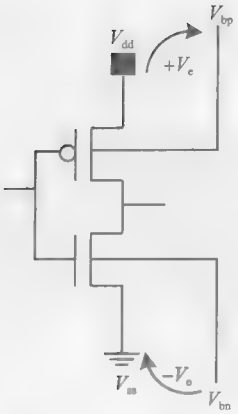


图 5.33 衬底偏压

V_{th}的升高也会影响性能，因此可以使用动态偏置的方式，即在工作模式下保持较小的偏置电压，而在保持模式下加强偏置电压。衬底偏压的效果与工艺尺寸有关，所以在使用更小工艺时，衬底偏压的效果会大大降低。

5.7.4 减少氧化层厚度

栅氧化层是门栅和沟道之间的绝缘体，常常将它做得尽可能薄，以使晶体管在打开时将沟道的导电性和性能增至最大，并在晶体管关闭时将阈限下漏电流降到最低。然而，在电流栅氧化层厚度在 1.2nm 附近时，会在门栅和沟道之间发生电子隧道效应的量子力学现象，使功耗增加[95]。绝缘体比硅氧化物有更大的介电常数（称为高 k 介电质），如混合型金属硅酸盐。如在 45nm 以及更小工艺中使用铪和锆硅酸盐氧化物来减少栅漏电流[95]。

5.7.5 多氧化层器件

多氧化层器件与 5.7.4 节中的描述很相似，除了厚氧化层头部/尾部开关也用来降低栅漏电流。

5.7.6 利用定制设计减小电容

下面公式中的 C_{out}为三种电容之和：

$$C_{out} = C_{fo} + C_w + C_p$$

式中 C_{fo} = 扇出门的输入电容

C_w = 线电容

C_p = 寄生电容

对于深亚微米技术，C_w 是最主要的一种电容，遗憾的是，它也是最难估计的一种电容。必须将“串扰”之类复杂效应考虑在内。

由于设计者很难对门级以下的单元进行布局布线，因此对这些单元的参数无法产生重要影响。而使用定制设计就可以对这样的参数进行强有力的控制。

参考文献

1. Application Note: AN1504/D, Metastability and the ECLinPS™ family, ON Semiconductor, Nov 2004, Rev 0.2
2. Metastability, Lecture at 5th Prague summer school on mathematical statistical physics, 2006
3. Rosenberger FU (Apr 2001) Metastability. EEEE463, Washington University, Electrical Engineering
4. Technical Note TN1055, Metastability in lattice devices, Lattice Semiconductor, Mar 2004
5. Application Note, Metastability characterization report for Actel flash FPGAs, Actel Corporation, July 2006
6. Alfke P (2005) Metastable recovery in Virtex-II Pro FPGAs. Xilinx, 10 Feb 2005
7. Stephenson J (2005) Design guidelines for optimal results in FPGAs, Altera Corporation
8. Wikipedia MOSFET and equivalent oxide thickness, free encyclopedia

第6章

流水线的艺术

6.1 介绍

对高速 ASIC 日益增长的需求使得越来越需要增加电路每个时钟周期的计算吞吐率。可以通过流水线提高 ASIC 在这方面的性能，但是也会带来系统延迟和面积的增加。

流水线通过在较长的组合逻辑路径中插入寄存器降低了组合逻辑的延迟，从而增加了时钟频率并提高了性能。

图 6.1 为插入流水线前的组合逻辑电路。图 6.2 为插入流水线后的电路示意图。

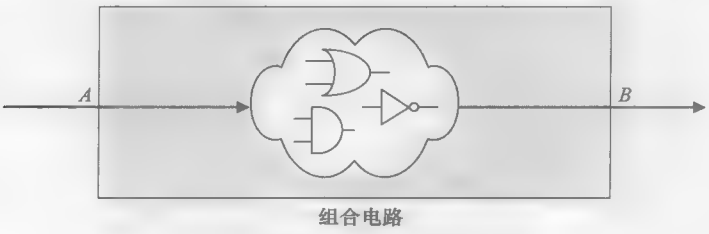


图 6.1　非流水线电路中的组合路径

图 6.1 中的组合路径延迟为 X 个时间单位（在 A 点和 B 点之间）。同样的路径在图 6.2 中通过插入三个寄存器被分割成多个小块，寄存器间的延迟为 Y 个时间单位，这里 $Y < X$。

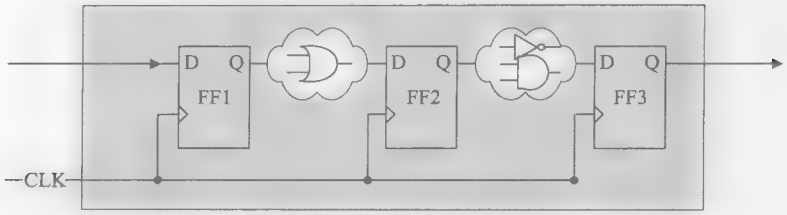


图 6.2 流水线电路中相对较小的延迟

从图 6.2 中能很明显地看出在长组合路径中加入三个寄存器之后时钟频率明显地增加了，但是同时也增加了额外的开销，并且增加了系统延迟。许多因素都会影响最大时钟频率。6.2 节在深入介绍流水线之前，简要地阐述这一内容。

6.2 影响最大时钟频率的因素

时钟频率是数据流入系统后在输出端出现的速率。有许多因素都会影响流水线系统的最大时钟频率。首先，考虑图 6.3 中所示的两条流水线阶段之间的理想路径。

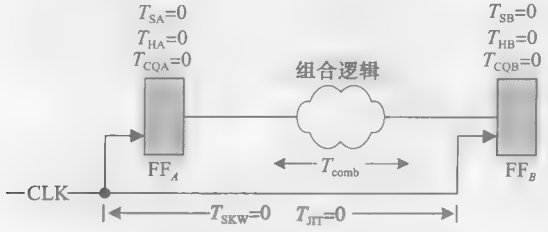


图 6.3 寄存器到寄存器路径间的理想状况

这里

T_{comb}：寄存器 *A* 和 *B* 间的组合延迟

T_{SA}：寄存器 *A* 的建立时间

T_{HA}：寄存器 *A* 的保持时间

T_{CQA}：寄存器 *A* 的时钟输出延迟

T_{SB}：寄存器 *B* 的建立时间

T_{HB}：寄存器 *B* 的保持时间

T_{CQB}：寄存器 *B* 的时钟输出延迟

对于没有任何抖动的理想时钟，时钟信号同时到达两个寄存器块，假设寄存器 A 时钟到输出的延迟（T_{CQA}）为0，寄存器 B 的数据建立保持时间（分别为 T_{SB} 和 T_{HB}）也为0，最大时钟频率 F_{max} 是通过组合逻辑的最大路径延迟的倒数，即：

$$F_{max} = 1/T_{period} = 1/T_{comb}$$

在实际的电路中，有许多其他的因素也会对时钟频率产生影响，如时钟偏移和抖动。下面将会加以讨论。

6.2.1 时钟偏移

在实际电路中，由于存在线路上的传播延迟，寄存器 B 的时钟输入（参考图6.3）相对于寄存器 A 可能会有一些延迟。

这些传播延迟中的微小差别，会发生在复杂数字产品的整个时钟网络上，最后会对整系统时序产生无法接受的影响。这种现象也称为“时钟偏移”问题[3]。

在相邻两个寄存器的时钟延迟大于这两个寄存器之间的数据路径延迟时，就会产生负时钟偏移。在这种情况下，先到的时钟会引起竞争条件。即，在数据还未成功锁存时时钟就触发了寄存器[2]。时钟偏移倾向于增加电路所能承受的最大时钟频率[3]。

6.2.2 时钟抖动

到达电路中某一点的连续时钟边沿之间间隔的变化称为时钟抖动 t_{jit}。

如图6.4所示，时钟抖动会影响时钟的占空比。让我们看看在实时电路中上述因素对最大时钟频率的影响。图6.5是典型的组合电路路径。粗体显示的路径（***b***、***f***、***j***、***i***、***m***、***n*** 和 ***o***）是两个寄存器之间最大延迟的路径。

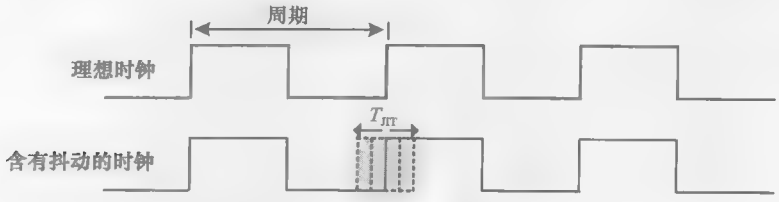


图6.4 时钟工作周期内抖动的影响

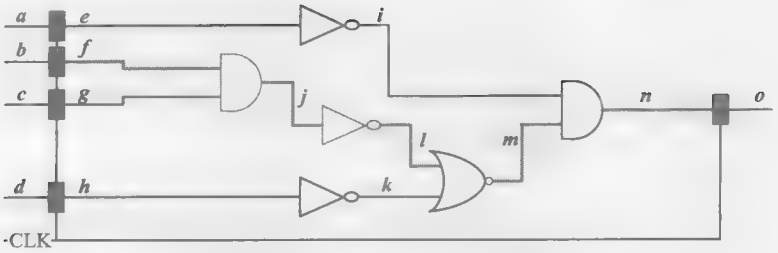


图 6.5 粗体显示带有最大组合延迟的关键路径

注意

路径 *c*、*g*、*j*、*l*、*m*、*n*、*o* 和 *b*、*f*、*j*、*l*、*m*、*n*、*o* 的延迟相同，但是这里在计算最大频率时只使用了后者。

让我们算一下从寄存器“*bf*”到输出“*o*”之间精确的组合延迟。以上路径的时序图见图 6.6。

图 6.6 各时钟源时钟对延迟贡献的时序图

T_{CQ}：寄存器“*bf*”的时钟输出延迟

T_{AND}：与门延迟

T_{INV}：反向器延迟

T_{NOR}：或非门延迟

T_{stp}：触发器的建立时间（对于图 6.5 中输出端的触发器）

$T_{SKW} + T_{JIT}$：时钟偏移和时钟抖动对组合延迟的总贡献

按照图 6.6，两个寄存器之间的路径 *b*、*f*、*j*、*l*、*m*、*n*、*o* 的总延迟为：

$$T_{FF} = T_{CQ} + T_{AND} + T_{INV} + T_{NOR} + T_{stp} + T_{SKW} + T_{JIT}$$

$$T_{FF} = T_{CQ} + T_{combo} + T_{stp} + T_{SKW} + T_{JIT}$$

因此对于给定电路使用下面的公式来计算最大周期：

$$\{T_{FF}\}_{max} = \{T_{CQ} + T_{combo} + T_{stp} + T_{SKW} + T_{JIT}\}_{max}$$

假设设计中所有触发器的延迟相等（实际情况不可能这样），那么得到：

$$\{T_{FF}\}_{max} = T_{CQ} + T_{stp} + T_{SKW} + T_{JIT} + \{T_{combo}\}_{max} \tag{6.1}$$

上面等式中的组合延迟部分可以通过添加多个触发器来减少，因此增加了电路操作的最大频率。这种减少流水线各阶段组合延迟的方法能显著提升电路的吞吐率（计算并未完成），后续几节将进行详细的介绍。

6.3 流水线

流水线使用存储器件将时钟周期内关键路径（最大组合延迟的路径）分割开来。这减少了关键路径上各阶段延迟并使电路能以更高频率工作。流水线电路增加了各时钟阶段的计算能力，但是由于使用了存储器单元也增加了负载。让我们来看一下图 6.7 中执行这种操作的电路：

$$i = (a + b + c + d) + (e + f + g + h)$$

计算图 6.7 中两个触发器之间的延迟（最大路径延迟）。

从式（6.1）

$$\{T_{FF}\}_{max} = T_{CQ} + T_{stp} + T_{SKW} + T_{JIT} + \{T_{combo}\}_{max}$$

这里 $\{T_{combo}\}_{max} = 3T_{adder}$

所以最终的时钟周期为

$$\{T_{FF}\}_{max} = T_{CQ} + T_{stp} + T_{SKW} + T_{JIT} + \{3T_{adder}\} \tag{6.2}$$

假设有下列约束值

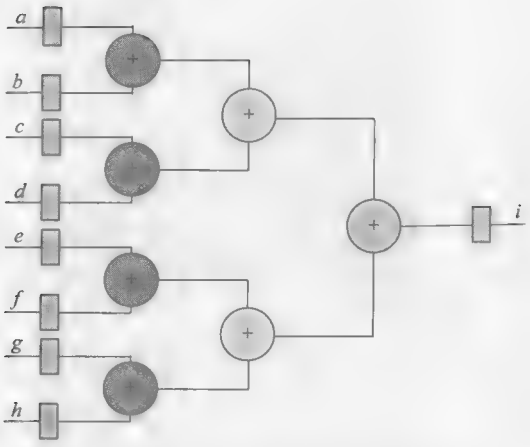


图 6.7 插入流水线前的 8 输入加法器

$$T_{CQ} = 4FO4$$

$$T_{stp} = 2FO4$$

$$T_{SKW} + T_{JIT} = 4FO4$$

$$T_{adder} = 10FO4$$

这里 FO4 指 4 个反向器延迟的扇出。

将这些值代入式（6.2）中，得到

$$\{T_{FF}\}_{max} = 4 + 2 + 4 + 3 \times 10$$

$$= 40FO4$$

现在让我们看一下同样的电路插入两级流水后的样子。新的流水线电路如图 6.8 所示。第一级流水线寄存器加在第一次加法操作之后，在

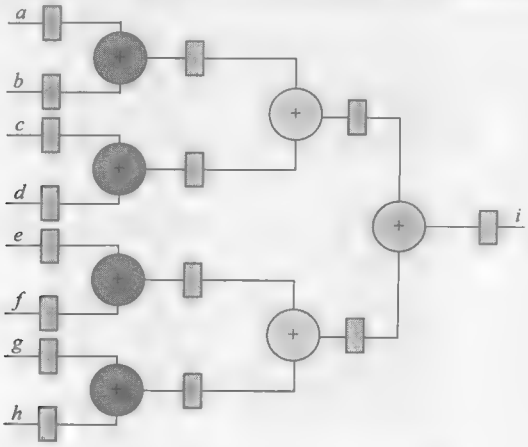


图 6.8 插入流水线后的 8 输入加法器

图 6.8 中表示为灰色，其他流水线寄存器加在各阶段直至加法器最终的输出被锁存住。

各级流水状态如下所示：

$$\{T_{FF}\}_{max} = T_{CQ} + T_{stp} + T_{SKW} + T_{JIT} + \{T_{combo}\}_{max}$$

其中 $\{T_{combo}\}_{max} = 1T_{adder}$

注意

在任何两个寄存器之间现在只存在一个加法器，而不再是三个加法器。

$$\{T_{FF}\}_{max} = T_{CQ} + T_{stp} + T_{SKW} + T_{JIT} + \{1T_{adder}\}$$
$$\{T_{FF}\}_{max} = 4 + 2 + 4 + 1 \times 10$$
$$= 20FO4$$

在使用 7 个流水化加法器实现 8 输入加法之后（如图 6.8 所示），吞吐率（每个周期的计算次数）增加至每个时钟周期计算一次 8 输入之和。总延迟为 3 个时钟周期。

与使用单加法器进行上述计算相比，7 个加法器意味着至少 7 倍的面积和功耗开销。将电路并行化对功耗和面积的影响都很大。一般来说，使用并行电路进行同样的 k 次操作，比重复使用某一逻辑 k 次达到同样的效果在面积和功耗方面的开支更大，因为使用了更多的触发器和额外逻辑，导致了更多的连线。

6.4 解释流水线——一个真实的例子

在讨论 CPU 技术时经常会出现“流水线”这个词，但是很少看到它的定义。尽管流水线的含义很简单，本章仍会用类比的方式来解释它的工作原理。让我们来看看汽车生产工序中的各阶段。

阶段 1：建造底盘；

阶段 2：将引擎放入底盘；

阶段 3：安装车门、外壳并罩住底盘；

阶段 4：安装轮子；

阶段 5：给汽车上漆。

如果在汽车制造流程中使用流水生产线，最好使用 5 组专业人员，每个阶段一组。一组建造底盘，一组建造和安装引擎，另一组安装轮子等。

完成每个阶段需要1个小时。这就是流水生产线的工作原理。

将所有5组人员排成一行，第一组人员开始第1阶段工作。第一阶段工作完成后，将汽车沿着生产线送到下一个阶段，由下一组人员放入引擎。在第2阶段人员向底盘中安装引擎时，第一组人员（以及所有其他阶段的人）是空闲的。第2阶段完成后汽车被送至第3个阶段，由第三组人员接管工作，而第二组人员进入空闲状态。

按这种方式汽车沿着生产线向下流动，每次只有某一阶段的人员处于工作状态，而其他人都是空闲的。在第5个阶段完成时，第1个阶段开始生产另一辆汽车。按这种速度制造一辆汽车要5个小时，即工厂每5小时才能送出一辆汽车。

另一种组织方式是雇用一个专职团队完成全部工作。由于组装汽车需要5种特定的技能，如果雇用5个高度专业化的团队来完所有工作，比雇用一个团队能达到更快的效果，因为一个团队很难对所有5个阶段的工作都很擅长。

如果适当地调整各团队的时间安排表和工作流程，每个小时就可以产出一辆汽车，这便极大地提高了装配线的生产效率。

第一组做好底盘后交给第二组，在第二组安装引擎的同时，第一组开始制作另一个底盘。

在流水线满负荷运转时，5组人员同时工作，这样每小时就能生产出一辆汽车了：即将生产效率提高了5倍。这就是流水生产线的样子，简单来说，就是流水线。

所以，回到数字设计的世界中，下一节会介绍大量提高数据设计性能的详细方法。

6.5 来自于流水线的性能提高

如图6.9所示，在两个寄存器之间存在一个大规模组合逻辑矩阵。

流水线的延迟是从数据进入流水线的输入端到这些数据经过处理后从流水线的输出端输出所消耗的时间。

图6.9中所示的电路只有一个流水线阶段（也称为非流水线阶段）。整个电路的延迟即时钟周期。

$$T_{\text{latency}} = T_{\text{comb}} + T_{\text{register}} + T_{\text{clocking}} \tag{6.3}$$

这里 $T_{register}$ 是寄存器开支 $= T_{CQ} + T_{stp}$。

$T_{clocking}$ 是时钟开支 $= T_{SKW} + T_{JIT}$。

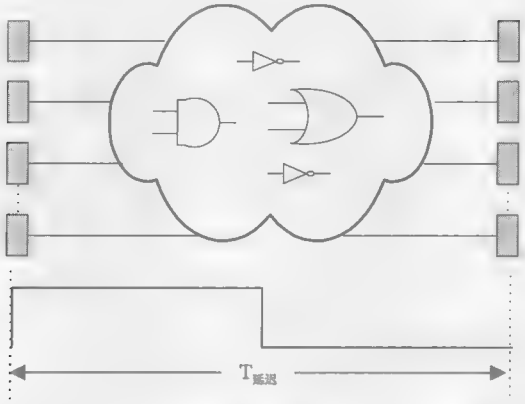


图 6.9 插入流水线前的逻辑（非流水线电路）

图 6.10 中是把同样的电路插入 n 级流水线的结果。

任意阶段的周期为：

$$T_{stage} = (T_{comb})_{stage} + T_{register} + T_{clocking}$$

延迟最大的流水线阶段决定时钟周期，对于任意阶段：

$$T_{pipeline} = \max\{T_{comb}\} + T_{register} + T_{clocking}$$

延迟是时钟周期的 n 倍，因为各阶段的延迟就是时钟周期。

$$T_{latency} = nT_{pipeline}$$

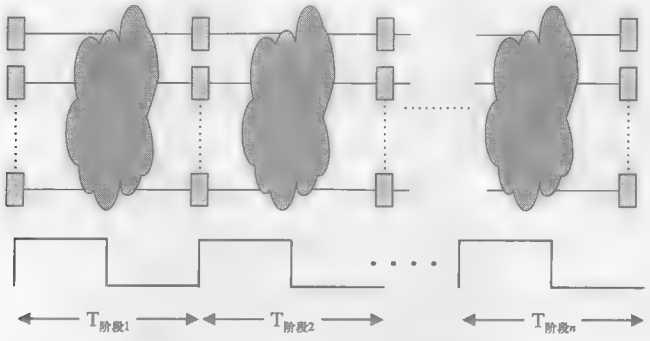


图 6.10 插入 n 级流水线后的逻辑

理想状况下，各级流水线的延迟应当相等，以使流水线中最大的组合逻辑延迟相同。

$$T_{\text{comb}_i} = \frac{T_{\text{comb}}}{n} \tag{6.4}$$

所以对任意流水线阶段，最小可能的时钟周期为：

$$T_{\text{pipeline}} = \frac{T_{\text{comb}}}{n} + T_{\text{register}} + T_{\text{clocking}} \tag{6.5}$$

因此对于理想时钟最终的延迟为：

$$T_{\text{pipeline\_ideal}} = nT_{\text{pipeline}} = T_{\text{comb}} + n(T_{\text{register}} + T_{\text{clocking}}) \tag{6.6}$$

现在就能计算出插入流水线后电路速度的增加了：

$$\text{速度增加} = \frac{F_{\text{after}}}{F_{\text{before}}} \tag{6.7}$$

这里 F_{after} 插入流水线后的时钟频率，F_{before} 是插入流水线前的时钟频率。

$$\text{速度增加} = \frac{T_{\text{before}}}{T_{\text{after}}} \tag{6.8}$$

从式（6.3）和式（6.5）可以得到

$$\text{速度增加} = \frac{T_{\text{comb}} + T_{\text{register}} + T_{\text{clocking}}}{\left(\frac{T_{\text{comb}}}{n} + T_{\text{register}} + T_{\text{clocking}}\right)}$$

对未插入流水线的电路，设寄存器和时钟开支占总时钟周期的百分比为 k，则：

$$k = \frac{T_{\text{register}} + T_{\text{clocking}}}{T_{\text{comb}} + T_{\text{register}} + T_{\text{clocking}}} \tag{6.9}$$

将 k 代入式（6.8）得到

$$\text{速度增加} = \frac{1}{\left(\frac{1-k}{n}\right) + k} \tag{6.10}$$

系统吞吐率是每个时钟内完成的计算量。因此流水线系统的性能可以定义为：

$$\text{流水线系统的性能} = \frac{\text{插入流水线前每个指令平均运算时间}}{\text{插入流水线后每个指令平均运算时间}}$$

设每个时钟周期 T 执行的指数为 IPC，

则每条指令的执行时间为 T/IPC，

代入上面的值，得到性能的增加：

$$\text{性能增加} = \frac{IPC_{\text{after}} \times T_{\text{before}}}{IPC_{\text{before}} \times T_{\text{after}}} \tag{6.11}$$

$$= \frac{IPC_{\text{after}}}{IPC_{\text{before}}} \frac{1}{\left(\frac{1-k}{n}\right) + k} \tag{6.12}$$

以上方式假设 $IPC_{after} \leqslant IPC_{before}$，并且在对电路插入流水线后不再使用任何微结构特性来提高 *IPC*。

让我们通过一个实际的例子来理解流水线对性能的提高：

从 Pentium III 到 Pentium IV，流水线级数从 10 增加到 20，使每周期指令数（*IPC*）减少了 20%。假设相对于非流水线电路会衡定增加 2% 的时序开支，通过下面的方式来计算性能增加：

$$\text{性能增加} = \frac{IPC_{after}}{IPC_{before}} \frac{\left(\frac{1-k}{n_{before}}\right)+k}{\left(\frac{1-k}{n_{after}}\right)+k}$$

$$= 0.8 \times \frac{\left(\frac{1-0.02}{10}\right)+0.02}{\left(\frac{1-0.02}{20}\right)+0.02}$$

$$= 1.37$$

可以看到，在同样的工艺下，尽管流水线级数增加了 2 倍，但 Pentium IV 仅仅比 Pentium III 提高了 37% 的性能。

注意

在 Pentium IV 中 IPC 减少 20%，是由于错误的分支预测，流水线停止和与 Pentium III 相比高度复杂的逻加所导致的。

让我们看看 DLX 微处理器中指令流程的实际例子。

6.6 DLX 指令集的实现

DLX 是新兴学院派标准结构的理论 32 位 RISC 微处理器。每条 DLX 指令最多由 5 个部分组成。

指令获取（Instruction Fetch，IF）、指令解码（Instruction Decode，ID）、执行/有效地址周期（Execution/Effective Address Cycle，EX）、存储器访问（Memory Access，MEM），和写回操作（Write Back，WB）。

未插入流水线的实现方式并不能达到最经济和最高的性能。所以很自然会使用流水线设计实现。

每个 DLX 指令最多用 5 个时钟周期完成。这 5 个时钟周期为：

1. 指令获取（IF）

IR <= MEM[PC]

$$NPC <= PC + 4$$

操作：

- 从存储器中获取指令（用 PC 指针）并放入指令寄存器（IR）。
- IR 保存下个时钟周期所需指令。
- PC 值递增 4，指向下一个指令地址。

2. 指令解码/寄存器获取（ID）

$$A <= Reg[IR_{6..10}]$$

$$B <= Reg[IR_{11..15}]$$

$$IMM <= \{[IR_{16}]_{16} IR_{16..31}\}$$

操作：

- 分析 IR 中的指令并访问寄存器堆以读取寄存器。
- 将通用寄存器的输出读入两个临时寄存器 *A* 和 *B*，供以后使用。
- IR 的高 16 位经过符号扩展保存到临时寄存器 IMM 中供以后使用。
- 由于指令格式是固定的，因此读寄存器和解码可以并行进行（域 $[IR_{0..5}]$ 指定操作类型）。这称为固定域译码。

3. 执行/有效地址周期（EX）

ALU 对上一个时钟周期准备好的操作数进行操作，根据 DLX 指令的类型执行下面 4 个功能中的一个：

a）访问存储器

$$ALUoutput <= A + IMM$$

操作：

- ALU 通过加法运算形成有效地址，并将结果放入寄存器 ALUoutput 中。

b）寄存器—寄存器 ALU 指令

$$ALUoutput <= A \ op \ B$$

操作：

- ALU 根据操作码对寄存器 *A* 和寄存器 *B* 中的数值进行操作，把结果放在临时寄存器 ALUoutput 中。

c）寄存器—立即数 ALU 指令

$$ALUoutput <= A \ op \ IMM$$

操作：

- ALU 根据操作码对寄存器 A 和寄存器 IMM 中存放的值进行操作，把结果放入临时寄存器 ALUoutput 中。

d）分支指令

$$ALUoutput <= NPC + IMM$$

$$Cond <= (A\ op\ 0)$$

操作：

- ALU 将 NPC 和 IMM 中的带符号立即数相加，计算出分支的目标地址。
- 检查寄存器 A（在前一个周期读取）的值来决定是否进行分支。比较操作 op 是由分支操作码所决定的关系操作符。

4. 访问存储器/分支完成周期（MEM）

这个周期里有效的 DLX 指令只有载入、存储和分支

a）访问存储器

LMD <= Mem[ALUoutput] 或 Mem[ALUoutput] <= B

操作：

- 如果指令为载入操作，则把从存储器中返回的数值存入 LMD 寄存器中。
- 如果指令为存储，则将寄存器 *B* 的值写入存储器。

注意

ALUoutput 是 ALU 阶段寄存器的输出。

b）分支

```
If (cond)
  PC <= ALUoutput
Else
  PC <= NPU
```

操作：

- 如果指令分支，就用 ALUoutput 寄存器中的分支目标地址替代 PC 中的值，否则用 NPC 寄存器中递增的值替代 PC 中的值。

5. 写回周期（WB）

a）寄存器—寄存器 ALU 指令

$$Reg[IR_{16..20}] <= ALUoutput$$

b）寄存器—立即数周期

$$Reg[IR_{11..15}] <= ALUoutput$$

c）取指令

$$Reg[IR_{11..15}] <= LMD$$

操作：

- 以上操作将结果写回到寄存器堆中，结果可能来自于存储器（LMD）或来自于ALU（ALUoutput）。

图6.11是无流水线时DLX数据通路上的5个步骤（IF、ID、EX、MEM、WB）。

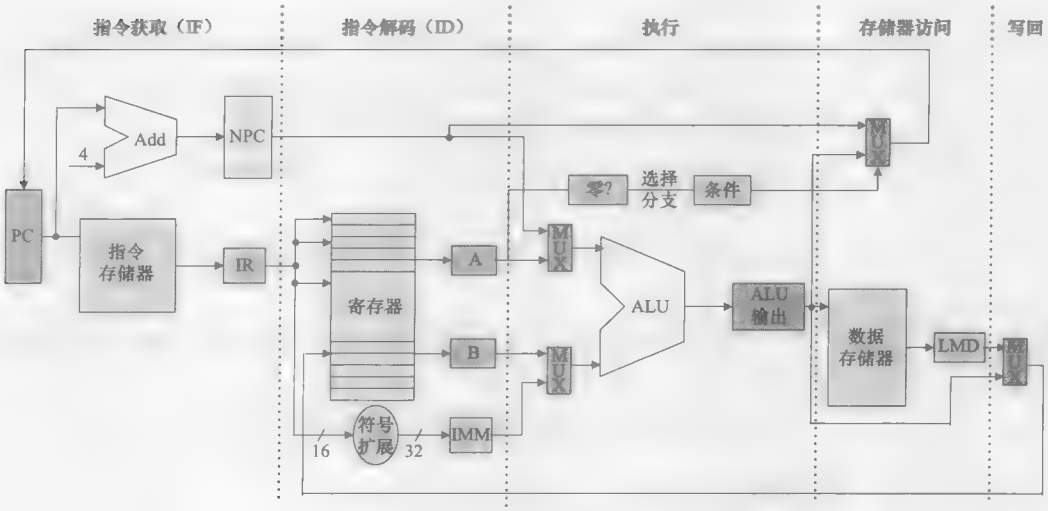


图6.11 单周期DLX数据路径

这里无法并行执行指令。只有在第一条指令执行完成后才开始执行第二条指令。

这种情况下执行一条指令需要8ns。指令按顺序一条接一条执行。所以4条指令的集合需要8×4=32ns才能执行完，见图6.12。

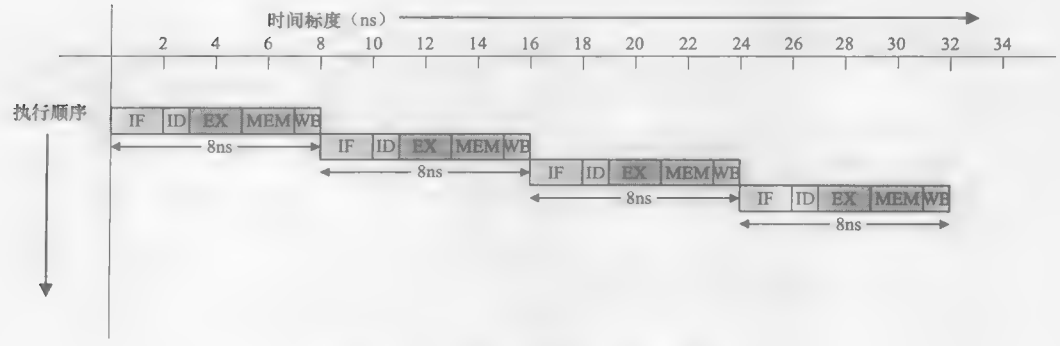


图6.12 非流水线系统中的执行顺序

6.7 流水线对吞吐率的影响

现在我们来对图6.11中的电路加以改造，对5级操作中的每个都加上一个流水线阶段（即在每个阶段加入一组寄存器）。修改后的新流水线电路如图6.13所示。

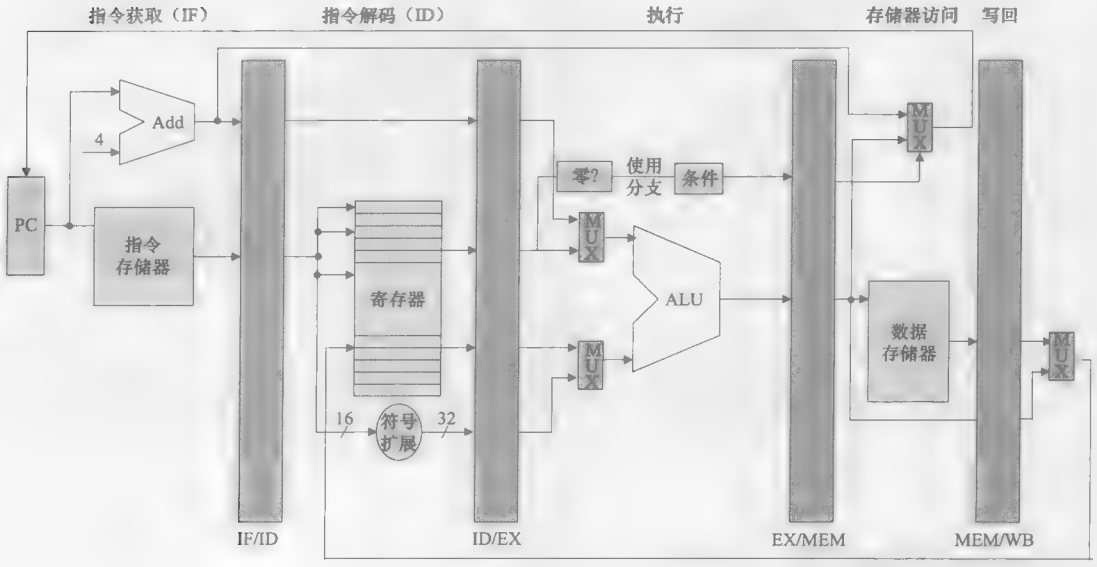


图6.13 使用流水线的DLX数据通路

每个时钟都获取一条指令并开始后序的5个执行周期（如图6.14所示）。

在图6.11中的原始单周期DLX数据通路中，指令无法并行执行，所以只有在前一条指令执行完成后才能处理新的指令。所以，假设单周期时钟周期是10ns，依次执行5条指令所用时间为5×10ns=50ns。

然后看看图6.14的多周期数据通路。此时每条指令需要5个时钟周期完成，但每个周期只需2ns。现在，如果5条指令像在单周期数据通路中那样连续执行，将花费5×5×2=50ns（即5条指令每条花费5个时钟周期，每个时钟周期为2ns）。然而，对于流水线模式中同样的数据通路，只需要5个时钟周期就可以执行完5条指令，如图6.6所示，也就是说，5条指令只需9×2ns=18ns就可完成。因此流水线使性能增加了50/18=2.8倍（对单个指令保持延迟值不变）。

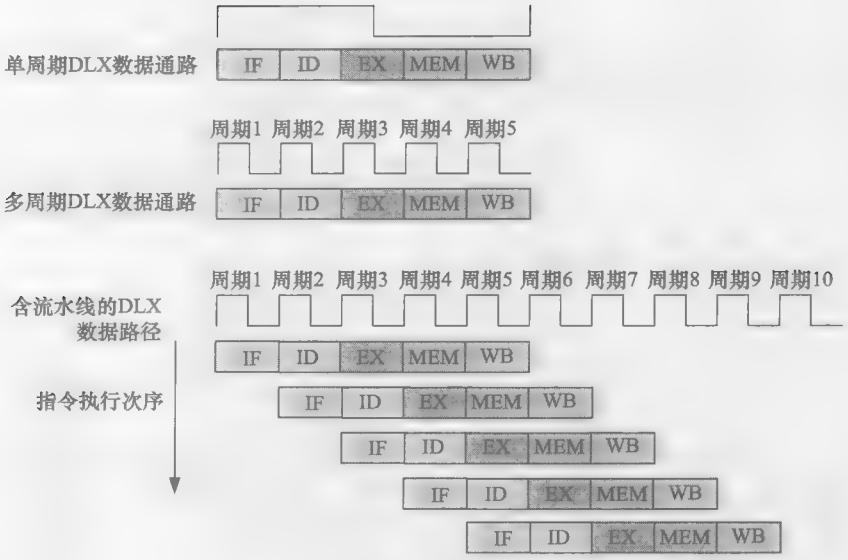


图 6.14 指令延迟和吞吐率

需要注意的是，在使用流水线时会引入额外的开销，如时钟偏移和寄存器延迟。这种开销（在图 6.14 没有表现出来）限制了所能达到的加速值。

6.8 流水线原理

- 所有共享一条流水线的指令的阶段和阶段次序必须相同。例如，“add”指令在存储器阶段中什么也不做。
- 所有中间值必须在各周期锁存。
- 不能复用任何功能模块。
- 一个阶段中的所有操作都应在一个周期内完成。

6.9 流水线冒险

冒险会干扰流水线并阻止下一条指令在目标时钟周期内的执行。冒险会降低流水线在理想情况下所能带来的速度提升。

冒险分类：

- 结构冒险　由于资源冲突而使硬件无法支持所有可能的指令组合同时执行。
- 数据冒险　指令执行需要之前指令的计算结果，而这个结果在流水线中还没有计算出来。
- 控制冒险　分支的流水线和其他指令改变程序计数器的值。

解决以上问题的通用方法是停止流水线直至风险消除，在流水线中插入多个“气泡”（缺口）。

6.9.1 结构冒险

在结构冒险中，硬件无法同时支持所有可能的指令组合重叠执行。结果会导致资源冲突，即多个指令要求访问同一地址。

例如，在图 6.15 中使用单端口存储器执行载入指令的操作。

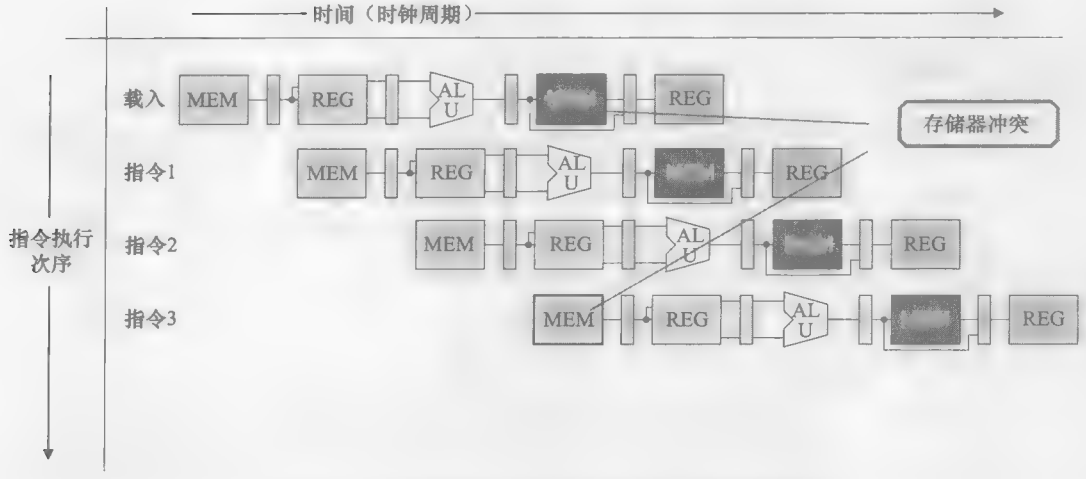


图 6.15　指令载入流水线过程中发生的存储器冲突

这里有两条指令（见图 6.15）在同一个时钟周期内要求访问同一存储器（一个在 MEM 阶段，另一个在 IF 阶段），因此就会产生存储器冲突。

上述问题的简单解决方法是在发生冲突时将流水线停一个时钟周期。这样会产生一个流水线气泡（如图 6.16 所示）。这样可以在增加一个时钟延迟的情况下执行同样数目的指令。

另一种解决方案是在 IF 和 MEM 阶段使用不同的存储器以避免同时访问同一块存储器的冲突。这样虽然解决了冲突冒险，但是消耗了更多的资源。

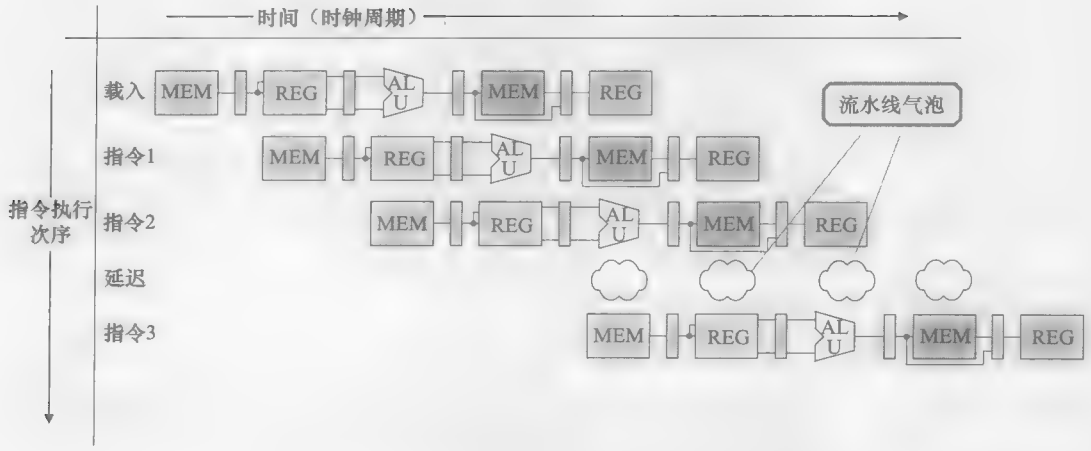


图 6.16 在流水线中加入气泡以阻止结构冒险

6.9.2 数据冒险

在数据冒险中，当前指令的执行需要依赖前面指令执行的结果。例如，对于下面的指令序列：

```
ADD   R1, R2, R3 (R1 = R1 + R2 + R3)
XOR   R4, R5, R1
SUB   R7, R1, R6
OR    R9, R8, R1
```

第一条指令后的所有指令都使用 R1（如图 6.17 所示）。

按图 6.17 中的方式，ADD 之后的指令无法得到正确的执行结果，因为在其指令周期的 ID 阶段 ADD 之后的所有指令的 ID 阶段都需要 ADD 指令的结果（在 WB 阶段输出）。

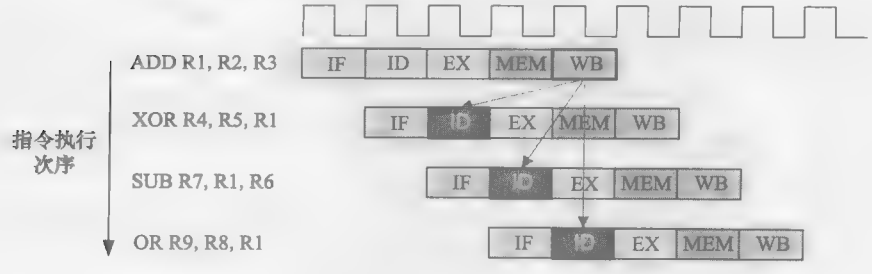


图 6.17 流水线指令集中的数据冒险

解决这种冒险的常用方法是数据/寄存器转移。其原理是选择的数据实际上并不在图 6.17 所示的 ID 阶段使用，而是在图 6.18 中的 ALU（EX）阶段使用。

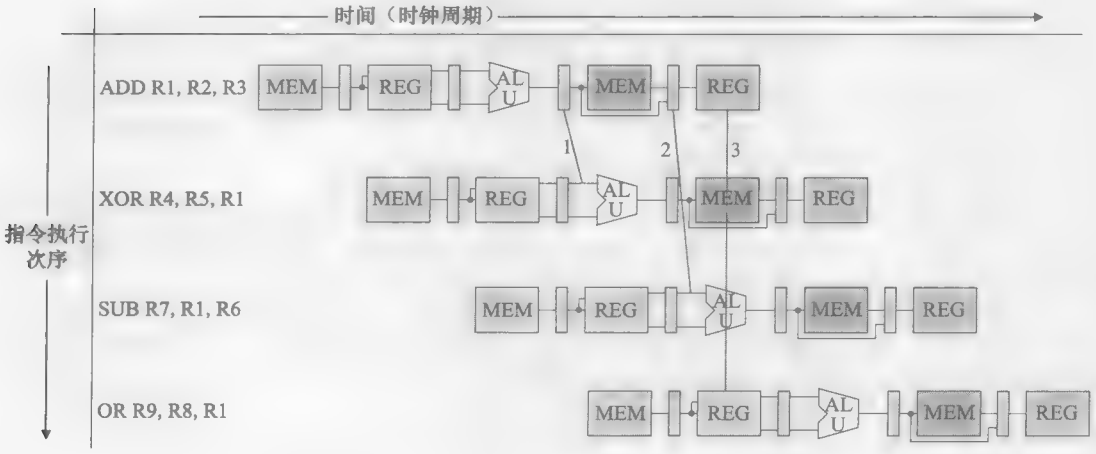


图 6.18 转移以避免数据冒险

数据转移规则如下：

- 第一条指令（该指令的输出会在后续指令中使用）的 EX/MEM 缓存输出的所有数据始终送到下一条指令的 ALU 输入端（如图 6.18 所示）。
- 如果转移硬件探测到源操作数有更新，逻辑电路会选择使用更新的结果，而不是来自于寄存器文件（ID/EX 缓存器）的数据。图 6.19 为使用多路器实现此操作的示意图。

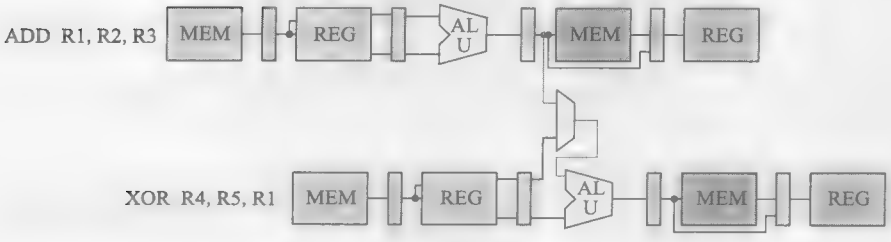


图 6.19 实现数据转移对硬件做出的改进

- 结果不仅需要从前一条指令转移，还需要从三个周期前开始的指令转移（如图 6.18 中的 1、2、3 所示）。EX/MEM（一个周期前）和 MEM/WB（两个周期前）的结果转移到两个 ALU 的输入

端（如图6.18中的1、2所示）。

- 在时钟周期的前半段读寄存器堆，在该时钟周期的前半段写寄存器堆（三个周期之前，如图6.18所示）。

存在影响电路得到正确结果的几种不同类型的数据冒险。

- 读后写（RAW）：这是最常见的数据冒险，可用数据转移解决。
- 写后写（WAW）：两条连续指令先后写同一寄存器，但写次序颠倒。DLX通过在WB阶段等待写入寄存器的方式避免这个问题。所以在DXL中不存在WAW风险。
- 写后读（WAR）：这里，在前一条指令读出寄存器的值（错误值）后，下一条指令才写入该寄存器。这种情况在DLX中也不会发生，因为所有指令会先读（在ID阶段）后写（在WB阶段）。

因此，数据转移不能解决所有类型的数据冒险。代表性的是直到MEM/WB阶段数据才能使用的情况。

例如，对于下面的指令：

```
LW   R1, 0 (R2)      LOAD 指令
XOR  R4, R5, R1
SUB  R7, R1, R6
OR   R9, R8, R1
```

如图6.20所示，路径1永远无法实现，因为XOR指令需要在ALU的输入端使用*R*1寄存器中的值，而该值在LW指令（从MEM/WB缓冲区）执行后才能更新。

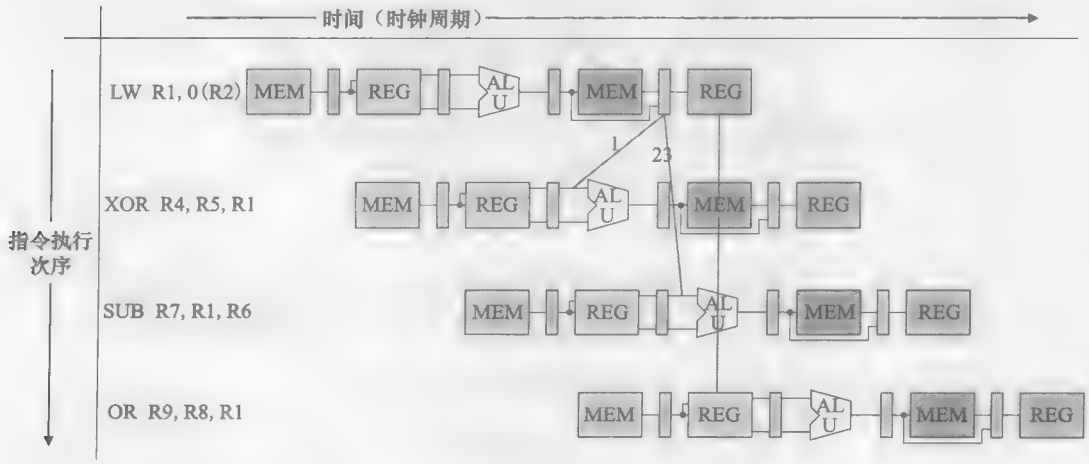


图6.20 使用转移仍会出现的数据冒险

一种简单的解决方法是使用如图 6.21 所示的“纵向气泡”将 ALU 周期延迟一个时钟。

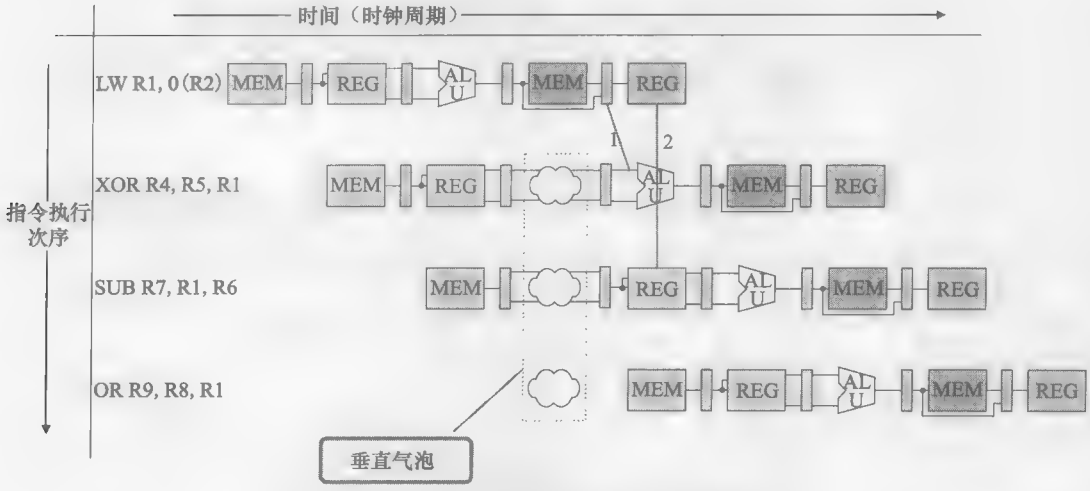


图 6.21 通过创建垂直气泡来解决数据冒险

除了使用硬件技术，也可以使用基于编译器调度的软件方式来解决数据冒险的问题。这时，编译器跟踪每个寄存器中的数据并重新安排指令次序以阻止数据冒险的产生。

6.9.3 控制冒险

这种冒险通常发生在由于分支语句使程序计数器（PC）发生变化的情况下。

例如，对于下面的代码段：

BNEZ R1，LOOP

DADD R4，R5，R6

如图 6.22 所示，在 T2 周期执行第二条指令时就需要 PC 的值，但是该值要在第一条指令的 MEM 操作（周期 T4）后才可用。

一种简单的解决方式是在分支语句中用新的 PC 值重新读取指令。在这种情况下，需要将流水线停止几个周期直到重新获取下一条指令，如图 6.23 所示。

通过提前预测分支目标或在分支延迟空隙间插入额外指令，可使流水线停止所导致的控制风险最小化。

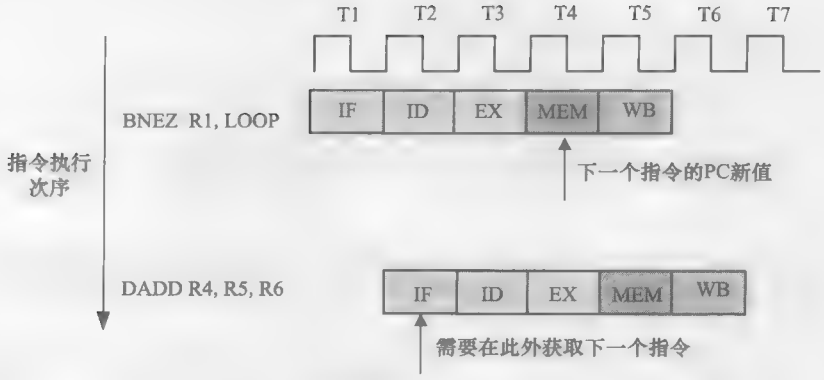


图 6.22 流水线化的指令组中的控制冒险

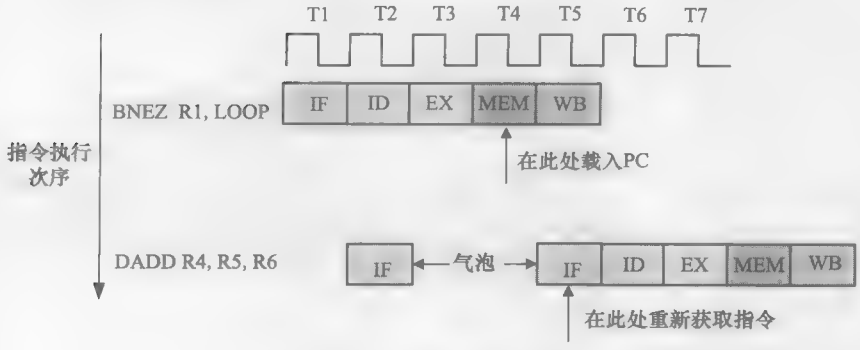


图 6.23 由于控制冒险停止流水线

6.9.4 其他风险

其他的一些风险也应该考虑。在指令获取或在数据读写时会访问存储器。在指令获取时，数据应该由 PC 寄存器保持稳定。该值应该一直保持到把获取的指令写入 IF/ID 流水线寄存器的 IR 域为止。因此，对 IF/ID 寄存器的 IR 写操作必须在 PC 写操作之前。

在数据读写时，访问存储器所用的有效地址在 EX 阶段由 ALU 计算出来。在将存储器中的数据存入 LMD 寄存器或存储器写信号被激活使数据写入存储器之前，该地址不能变化。

实际上，ALU 输出值与存储器访问是独立的过程，这就意味着有时 EX/MEM ALU 的输出值会在数据写入 MEM/WB LMD 寄存器前更新。需要

对这三个事件进行严格的排序以使 EX/MEM ALU 的输出只在存储器访问完成后才改变。注意，对于非流水线结构的 CPU 就不存在这样的问题，因为指令周期的访问阶段在执行阶段之后，而只有在执行阶段才会修改 EX/MEM ALU 输出的值。

6.10 ADC 中的流水线——一个例子

虽然你可能从未在处理器以外的其他结构中听到过流水线结构，但该理论可使用在任何设计中用于提高性能。

考虑下面的 ADC 例子。含有流水线的模数转换器（ADC）是最流行的 ADC 结构，其取样率可以从每秒几兆到每秒 100 兆以上[16]。

图 6.24 是 Maxim 的 12 位流水线 ADC 模块图，每个阶段处理两位。

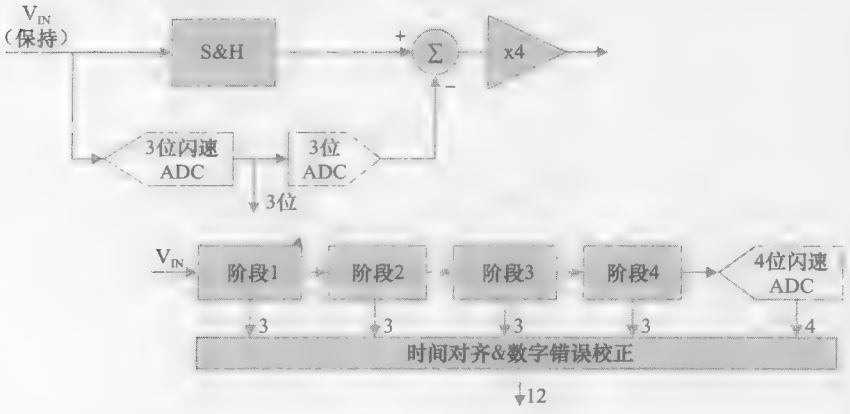


图 6.24 maxim 的 4 级 3 位流水线 ADC

图片来源：Copyright © Maxim Intergrated Products（http://maxim-ic.com），经授权使用

图 6.24 为位流水线 ADC 的结构图。输入 V_{IN} 首先被采样 - 保持（S&H）电路所采样，同时第一级的闪速 ADC 把它量化为 3 位，然后把此 3 位输出传递给一个 3 位的 DAC（具有 12 位精度），输入信号减去此 DAC 的输出，放大 4 倍传递给下一级（第二级）。继续重复上述过程，每级提供 3 位，直到它到达最后一级 4 位闪速 ADC，该 ADC 解析最后 4 个 LSB。对应某一次采样，由于每级在不同的时间得到变换结果，因此在进行数字误差校正前用移位寄存储器对各级的结果先按时间对准。

注意，只要某一级完成了某一采样的变换，得到结果并把差值传递给

下一级，它就可以开始处理从每一级的采样－保持电路中接收到的下一个采样。因此流水线操作提高了处理能力。

本节摘自 Maxim 使用手册 1023“理解流水线 ADCs”。

参考文献

1. Application Note AP-589, Design for EMI, Intel®, Feb 1999
2. Freescale semiconductor application note AN2764, Improving the transient immunity performance of microcontroller-based applications, www.freescale.com, 2005
3. Application Note 1023, Understanding pipelined ADCs, Maxim, 1 Mar 2001

第7章
处理字节顺序

7.1 介绍

字节顺序描述如何在计算机系统中表示由多个字节组成的数据。

例如，当传输单词“TEST”时，每个字母用一个包传输，一共4个包。传输端按下列次序发送数据“T（第一个发送）”→“E”→“S”→“T”（最后一个发送）。如果没有足够的信息，接收端可能以16种不同的组合方式捕获和组装数据。类似地，假设用两个包来传送该单词，每个包含两个字母（“TE”和“ST”），接收端可能将数据组装成“TEST”或“STTE”，后者是错误的。同理，当程序或数据在不同字节顺序的片上系统（SOC）间传输时就会面对这样的问题，除非所有计算机系统按同样的字节顺序设计。假设软件所访问的所有数据都是32位字，那么字节顺序的问题并不明显。然而，如果软件所执行的指令操作8位或16位数据，并且该数据需要映射到指定的存储器地址时（如存储器映射I/O），就需要处理字节顺序的问题了。

本章为芯片体系结构设计人员以及希望开发字节中性型代码或转换字节特定型代码的软件开发人员建立了一系列基本的规则。

7.2 定义

字节顺序定义数据在计算机系统中的存储格式。它描述存储器中地址的最高有效位（MSB）和最低有效位（LSB）的位置。对于数据始终以32

位形式保存在系统存储器中的真正32位系统，字节顺序没有什么实际意义，但是对于要将字节或16位半字映射到系统存储器中32位字的系统，字节顺序不匹配就会影响数据完整性。

字节顺序结构有两种类型，大端模式（Big Endian，BE）和小端模式（Little Endian，LE）。大端模式将MSB保存在最低存储器地址处。小端模式将LSB保存在最低存储器地址处。多字节数据的最低存储器地址可以认为是数据的起始地址。表7.1是分别按大端模式和小端模式保存在存储器中的32位十六进制数0xAABBCCDD。“字节0”表示最低存储器地址。

表7.1 大端和小端字节顺序

字节顺序结构	字节0	字节1	字节2	字节3
大端	AA（MSB）	BB	CC	DD（LSB）
小端	DD（LSB）	CC	BB	AA（MSB）

注意，在按原生数据类型（如32位）引用数据时，两种字节顺序所存储的多字节数据域是相同的。但在按字节或半字类型访问数据时，字域次序与系统的字节顺序配置相关。如果程序将表7.1中的各字以字的形式保存在0x100位置，并按独立字节形式取数，就可能存在两种次序。

在小端模式系统中，数据字节的顺序如表7.2所示。

表7.2 小端取址

地　址	数　据
0x0100	DD
0x0101	CC
0x0102	BB
0x0103	AA

注意，该字最右边的字节作为第一个字节保存在存储器中0x100的位置。这就是这种格式称为小端模式的原因；字的最低有效字节占据了存储器中该字的最低字节地址。

如果程序在大端模式系统中执行，表7.1中的字在存储器中的字次顺序如表7.3所示。

表7.3 大端取址

地　址	数　据
0x0100	AA
0x0101	BB
0x0102	CC
0x0103	DD

字的最低有效字节保存在高阶字节地址上。最高有效字节保存在低阶字节地址上，因此这种格式称为大端模式。

在以半字方式处理时，存储器地址必须是 2 的倍数。因此表 7.1 中的值对应两个半字地址：0x100 和 0x102。表 7.4 是在这种情况下全部字节顺序的配置方式：

表 7.4 半字字节顺序

地　址	小　端	大　端
0x0100	CCDD	AABB
0x0102	AABB	CCDD

注意

在半字中，字节保持与在字中一样的次序。在小端模式中，最低有效半字保存在低位地址（0x100）中，且最高有效半字保存在高位地址（0x102）中。对大端情况，分配方式正好相反。

通常字节顺序对程序员和用户是透明的。但是，在数据必须跨越字节顺序模式时往往会带来麻烦。

7.3 小端模式或大端模式：哪个更好

你可能遇到过很多关于两种格式优劣的争论，最多的争论发生在 PC 和 Mac 相对的优劣上；实际上两种格式都有其各自的优缺点。

在“小端”模式中，因为最低位字节偏移值为“0”，并首先访问，访问 1、2、4 或更长字节数的汇编语言指令能按同样方式处理所有格式。同样，由于地址偏移量和字节数之间 1:1 的关系（偏移 0 对应字节 0），多精度的数学程序也相对容易编写。

在“大端”模式中，因为先访问高阶字节，所以很容易通过查看偏移量为 0 的字节来判断数值的正负。所以无须接收所有字节包就能知道符号信息。由于数字同时也以其打印出来的次序保存，所以对从二进制转化为十进制的程序特别有效。

让我们看看在存储器中以不同形式的格式保存的十六进制数 0x12345678。

地　址	00	01	02	03
大端模式	12	34	56	78
小端模式	78	56	34	12

可以看出以大端模式读取和转储十六进制数很容易，因为通常从左向右读一个数（从较低地址到较高地址）。

大多数位图图像（显示和内存安排）映射成“MSB 在左侧”的方案，这可以使体系结构能够自然地移动和存储大于一个字节的图像元素。小端模式的性能在这种情况下处于劣势，因为在处理大型图像元素时需要将字节次序反向。

表 7.5 列出了一些流行的计算机系统的字节顺序结构。注意，某些 CPU 可以通过设置处理器寄存器而在大端或小端模式（双字节顺序）间切换。

表 7.5　计算机系统的字节顺序

处理器	字节顺序体系结构
ARM	双字节顺序
IBM Power PC	双字节顺序
Intel ® 80x86	小端模式
Intel ® Itanium ® processor family	双字节顺序
Motorola 68 K	大端模式

大多数嵌入式通信处理器和定制解决方案在数据层采用大端模式（即，PowerPC、SPARC 等）。因为为这些处理器所写的老程序通常遵循网络字节顺序（大端模式）。

常用文件格式和其字节顺序如表 7.6 所示：

表 7.6　通用文件格式及其字节顺序

文件格式	字节顺序格式
Adobe photoshop	大端模式
BMP（Windows and OS/2 Bitmaps）	小端模式
GIF	小端模式
JPEG	大端模式
PCX（PC Paintbrush）	小端模式
QTM（Quicktime Movies）	小端模式
Microsoft RIFF（. WAV &. AVI）	双字节顺序
Microsoft RTF（Rich Text Format）	小端模式
SGI（Silicon Graphics）	大端模式
TIFF	双字节顺序
XWD（X Window Dump）	双字节顺序

这意味着在将数字写入文件时，需要知道文件结构是怎样的，例如，如果在“大端模式”的机器上写入图像文件（如 BMP 文件），首先要将字节顺序反转，否则“标准的”程序在读该文件时会无法工作。

Windows 的 .BMP 在所有平台上坚持使用“小端模式”的格式，因为它是在“小端模式”体系结构上开发的。

此外注意，某些 CPU 可以通过设置处理器的控制寄存器切换为大端或小端（双字节顺序）模式。

7.4 处理字节顺序不匹配的问题

在单个系统中字节顺序不会产生什么问题。只有在两台电脑通信时它才会产生影响。每个处理器和每种通信协议必须选择一种字节顺序。因此，如果需要通过存储器通信，两种不同字节顺序类型的处理器就会产生冲突。类似地，小端模式处理器在试图访问大端模式网络时需要用软件对字节重新排序。

如果计算机无意间从共享存储器区域或文件中读取以相反格式写入的二进制数据时，字节顺序的差异就会导致问题。

另外一个字节顺序会导致问题的领域是网络通信。由于在同一网络中可能存在不同类型（大端模式和小端模式）的处理器，它们之间必须彼此通信。因此网络协议栈和通信协议也必须定义字节顺序。否则，不同字节顺序的两个节点可能无法通信。这是嵌入式程序员更常面对的情况。

事实上，TCP/IP 族中的所有协议层都定义为大端模式。也就是说，各层头中的 16 或 32 位值（如 IP 地址、包长度或校验和）必须首先发送或接收最高有效字节。

比如，你想与 IP 地址为 192.0.1.7 的计算机建立 TCP 套接字连接。IPv4 使用 32 位整数来识别网络中各主机。以上带点的 IP 地址必须转化为这样的一个整数。

TCP/IP 协议所使用的多字节整数表达方式有时称为“网络字节顺序”。即使各终端计算机都是小端模式的，它们之间所传输的数据必须在通过网络前先转化为网络字节顺序，然后在接收端再转化为小端模式。

假设一台基于 80x86 的小端模式 PC 与基于 SPARC 的大端服务器通过互联网通信。在没有进一步操作的情况下，8086 处理器将 192.0.1.7 转化为小端模式整数 0x070100c0，并按下列次序传输：0x07，0x01，0x00，

0xc0。SPARC 将按下列次序接收数据：0x07，0x01，0x00，0xc0，并将按大端模式重建整数 0x070100c0，并将地址误译成 7.1.0.192［7］。

阻止这种混乱的发生使得 TCP/IP 协议栈开发者需要处理令人厌烦的小细节的实现。如果栈在小端模式处理器上运行，就需要在运行时对各层头中每个多字节数据域重新排序。如果栈在大端模式机器上运行，就没有什么需要担心的了。对可移植的协议栈（即，能在两种处理器上运行），需要能对是否进行重新排序做出判断。通常在编译时进行该判断。

另一个好例子是对设备进行闪存编程。大多数常见的闪存存储器是 8 或 16 位宽的。大多数 32 位闪存存储器实际上要求两个交错的 16 位设备。对这些设备的编程操作涉及对各器件特定地址的 8 或 16 位数据写操作。因此，软件工程师必须知道和理解硬件的字节顺序配置，以便能成功对闪存器件编程。

在处理器从 8 或 16 位闪存设备中直接执行代码时，这些代码应该保存成能被处理器直接识别的形式。这会受到系统字节顺序配置的影响。编译器通常包含控制代码镜像的字节顺序的开关，以能把它正确编程到闪存设备中。

7.5 访问 32 位存储器

下面的例子是对 8 位、16 位和 32 位存储器的访问。

字节地址和 32 位数据总线上特定位的关系如表 7.7 所示。

表 7.7 不同字节顺序系统的地址数据映射

地址［1：0］	大端（BE）	小端（LE）
“00”	Data［31：24］	Data［7：0］
“01”	Data［23：16］	Data［15：8］
“10”	Data［15：8］	Data［23：16］
“11”	Data［7：0］	Data［31：24］

表 7.8 为按 8 位、16 位和 32 位访问大端与小端模式系统时的数据字节映射。

表 7.8 使用 8、16 和 32 位访问格式的不同字节顺序系统的地址数据映射

	数据［31：24］	数据［23：16］	数据［15：8］	数据［7：0］
数据［31：0］	0A	0B	0C	0D
字节地址（BE）	0	1	2	3
字节地址（LE）	3	2	1	0
32 位读操作				
地址“00”（BE）处的 32 位读操作	0A	0B	0C	0D

（续）

	数据[31: 24]	数据[23: 16]	数据[15: 8]	数据[7: 0]
地址“00”（LE）处的32位读操作	0A	0B	0C	0D
16位读操作				
地址“00”（BE）处的16位读操作	0A	0B	—	—
地址“00”（LE）处的16位读操作	—	—	0C	0D
地址“10”（BE）处的16位读操作	—	—	0C	0D
地址“10”（LE）处的16位读操作	0A	0B	—	—
8位读操作				
地址“00”（BE）处的8位读操作	0A	—	—	—
地址“00”（LE）处的8位读操作	—	—	—	0D
地址“01”（BE）处的8位读操作	—	0B	—	—
地址“01”（LE）处的8位读操作	—	—	0C	—
地址“10”（BE）处的8位读操作	—	—	0C	—
地址“10”（LE）处的8位读操作	—	0B	—	—
地址“11”（BE）处的8位读操作	—	—	—	0D
地址“11”（LE）处的8位读操作	0A	—	—	—

7.6 处理字节顺序不匹配

在包含若干IP的片上系统（SoC）中必然会发生字节顺序不匹配的问题，几乎所有第三方公司的IP都支持与处理器同样的字节顺序类型。处理字节顺序不匹配问题的最简单方法是为系统选择一种“字节顺序类型”（即，小端模式或大端模式），并将全部其他不同字节顺序的模块转化为目标“字节顺序类型”。

典型的字节顺序类型由系统中CPU体系结构的实现决定，所以强烈推荐目标的“字节顺序类型”与处理器的字节顺序类型相匹配。在对第三方IP选型时要考虑的另一个因素是确认其是否支持“双字节顺序”结构，以使IP可以方便地编程为“大端模式”或“小端模式”以与系统无缝集成。对于不满足这些要求的情况，必须使用本节提到的技术之一来解决字节顺序冲突的问题。万一没有可编程选项，就需要在将IP集成到SoC的过程中解决字节顺序不匹配问题。

有两种连接相反字节顺序外设的方法。根据应用的需要，要么选择将地址保持稳定（即，在地址不变处字节保持在同一地址）或者将位顺序保持稳定（在数据不变处地址改变）。

7.6.1 保持数据完整性（数据不变）

在SoC的内核或IP对单个或多字节域进行操作时，域左边为MSB，右边为LSB。即如果用16位域表示整数并进行加操作，会对LSB加1，并且所有进位从LSB（在右边）向MSB（在左边）进行。该操作对大端或小端地址体系结构是相同的。

这样产生的主要问题之一是如何将内核和其他不同字节顺序地址体系结构的IP混合使用。由于不同的字节顺序模式下多字节域具有不同的字节地址，因此，如果将多字节域作为单个条目进行操作，那么当其在各IP之间移动时，必须保留该条目的位次序。

对于跨字节边界的多位域也同样如此。比如，一个含16位控制寄存器编程模型的IP。如果控制寄存器位域［8：7］定义了控制域，那么要求对于所有访问控制寄存器的操作，要保持该16位之间的稳定关系，而不再与字节顺序有关。

为了理解匹配字节顺序以使数据位顺序完整这样一个过程，考虑由小端模式的外设接收连续帧并将接收到的数据通过DMA/CPU保存到系统存储器这样一个例子，这里存储器（系统RAM）和CPU/DMA都是大端模式。如图7.1所示，首先接收到连续帧的头部，然后是帧的其余部分。

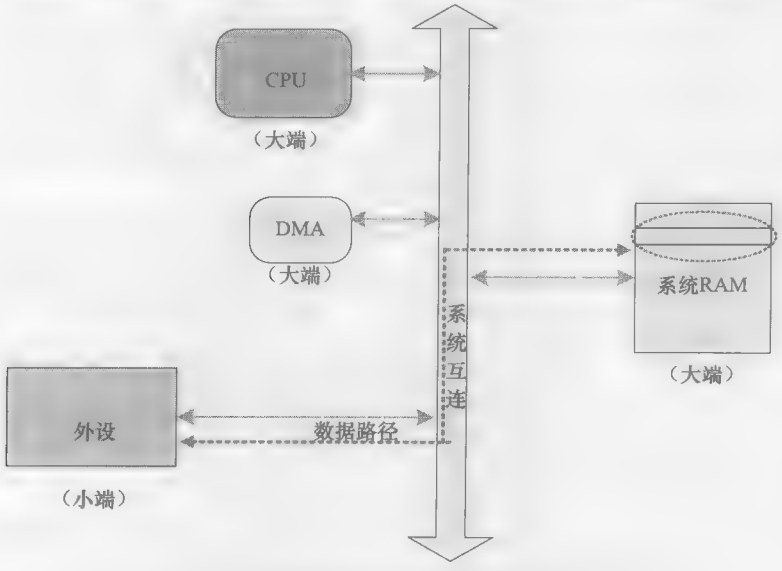


图7.1 从小端外设到系统存储器的数据流（地址变化）

接收到的连续帧以“Type”、“H2”、“H1”和“H0”的顺序保存在外设存储器中。帧中的域可能跨多个字节并不在字节边界处终止（见图7.2）。例如，状态域可能是12位。所以对于应用程序来说，数据不会因为字节顺序转换而改变就显得很重要，因为这样软件就可以以该顺序处理数据了。

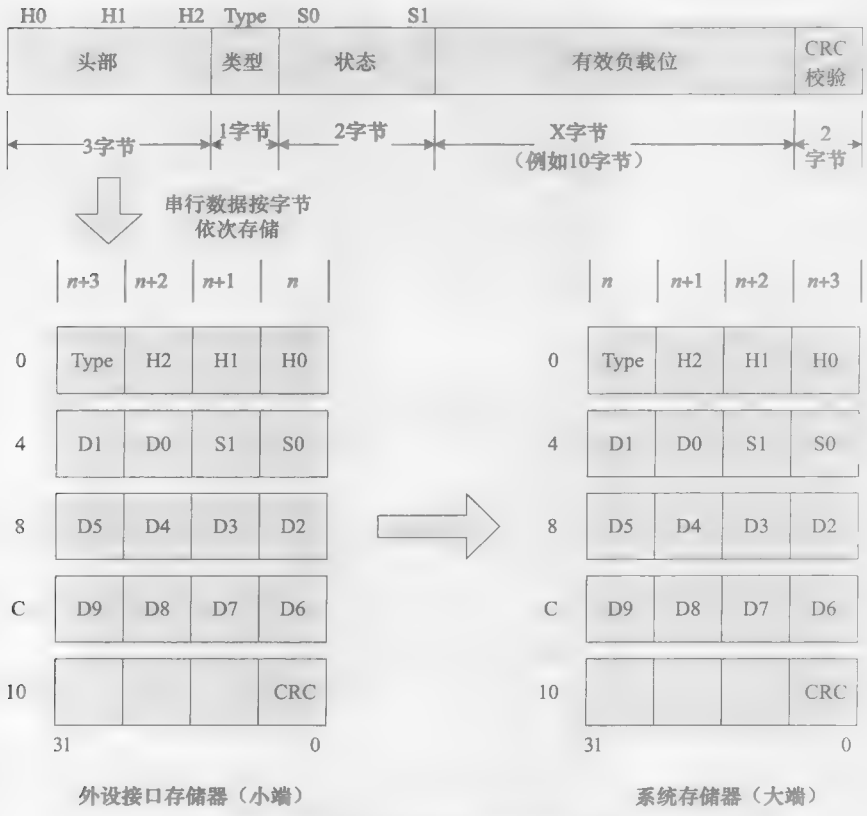


图7.2 使用数据不变性将小端存储器连接到大端存储器上

在图7.2中，数据按小端寻址保存在外设的存储器中。现在当数据要传送到大端模式的系统RAM中时，必须保证数据位顺序保持不变。为了用硬件达到这个目的，需要对访问外设RAM的存储器的地址进行修改对地址的改动基于要传送数据的大小，如表7.9所示。

表7.9 用以匹配字节顺序的地址变化

转换大小	小端地址	映射后的大端地址
8位	0x0003	0x0000
	0x0002	0x0001
	0x0001	0x0002
	0x0000	0x0003

（续）

转换大小	小端地址	映射后的大端地址
16 位	0x0002	0x0000
	0x0000	0x0002
32 位	0x0000	0x0000

使用上面的逻辑，对地址总线最后两个 LSB 取反，数据数据不变。图 7.3为对应的 HDL 代码。

```
assign le_ram_addr[ 31:0] = (Size = 8-bits)  ?
              { ram_addr[ 31:2] , ~ram_addr[ 1] , ~ram_addr[ 0] } :
                                  (size = 16-bits) ?
              { ram_addr[ 31:1] , ~ram_addr[ 0] , ~ram_addr;

assign le_ram_data[ 31:0] = data[ 31:0] ;
```

图 7.3 使用数据不变性对字节顺序匹配的代码

使用上面的方案可以使字节顺序转换对软件透明，并能确保数据完整性在字节顺序转换过程中不会被破坏。

数据流

从小端设备到大端存储器，并保持数据不变的数据流如下所述。

1）DMA 发出对外设存储器的读字节操作。

2）让我们举个例子，其中系统产生的地址为 0x00。在数据变化的实现中，小端模式设备 RAM 所看到的地址为 0x03。

3）设备存储器对该地址解码，访问位 31：24 或者如图 7.3 所示的“Type”域。

4）外设输出该值为｛“Type”，“0x000000”｝（32 位输出）。

5）DMA 对系统的大端存储器发出按字节方式的写操作。

6）再次产生 0x00 地址（字节访问）。

7）大端存储器将该访问解码为写入位 31：24。

8）由于来自于小端存储器的数据处于同样的字节区域，因此可以保持数据的完整性并将数据保存在大端 RAM 中。

9）对其他需要从外设 RAM 传输到系统 RAM 中的数据继续进行该操作。

10）16 位和 32 位访问过程与上面所述相同，只是地址需要加以改变，如表 7.9 所示。

7.6.2 地址不变

与数据不变的字节顺序转换相比，某些应用程序或系统不需要数据保

持特定的次序，但是需要在字节顺序转换后数据字节保持在同样的地址区域。这时就需要使用地址不变的字节顺序转换。

参考同一个接收连续帧的例子，对于地址不变的系统，访问字节“Type”的地址偏移量永远是0x3。而在前一节中，访问该字节需要使用不同的地址偏移量。为了用硬件实现该过程，需要修改或交换从外设RAM存储器中读到的数据值。数据的修改如图7.4所示。

```
assign le_ram_addr[31:0] = ram_addr;

assign le_ram_data[31:0] = data[0:31];
```

图7.4 使用地址不变性匹配字节顺序

地址不变的字节顺序转换如图7.5所示。

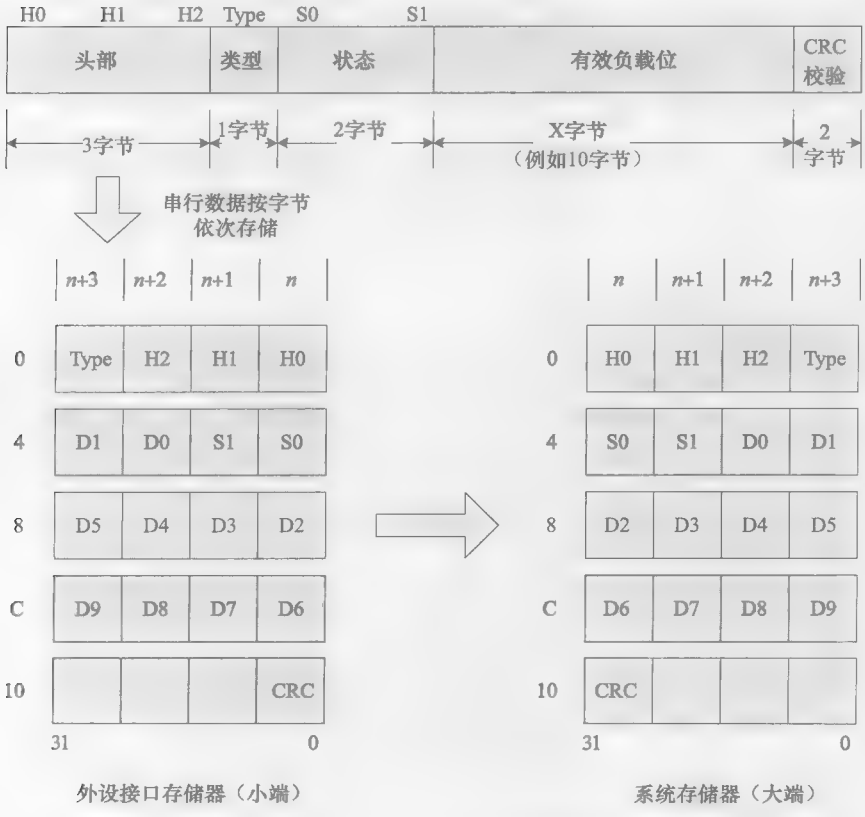


图7.5 使用地址不变性将小端存储器连接到大端存储器上

数据流

使用地址不变方式，从小端外设到大端存储器的数据流如下所述。

1）DMA 发出对外设存储器的读字节操作。

2）让我们举个例子，其中系统产生的地址为 0x00。地址不变的实现使地址值始终相同。

3）外设 RAM 对该地址解码，访问位 7：0 或“H0”区域，如图 7.5 所示。

4）外设输出的数据为｛“0x000000”，“H0”｝（32 位输出）。由于字节顺序匹配时地址不变，给系统 RAM 的数据变为｛“H0”，“0x000000”｝。

5）DMA 发出对系统大端存储器的字节写操作。

6）再次产生地址 0x00（字节访问）。

7）大端存储器将该访问解码为写入位 31：24.

8）在字节顺序转换完成后，从小端存储器读出的数据已处在同样的地址区域，把数据存入大端 RAM。

9）对其他需要从外设 RAM 传递到系统 RAM 的字节继续进行以上操作。

10）对 16 位或 32 位的访问，上述操作过程是相同的，只需按表 7.9 所示的方式将输出数据进行交换即可。

7.6.3 软件字节交换

交换字节是实现字节顺序转换的一种方式。在由应用程序本身决定字节顺序的系统中，该模式是有用的。因此，无须用硬件实现这种字节顺序匹配过程。字节顺序中性代码的字节交换方法使用字节交换控制来决定是否必须进行字节交换过程。

方法

在软件中通常使用的各种字节交换方法有：

- 交换汇编指令
- 用于交换字节的软件库宏
- 协议特定的交换函数
- 制定的交换函数

交换汇编指令

某些微控制器的指令集包含预定义的交换函数，软件可直接使用于实

现针对特定应用的字节顺序转换。

交换库宏

某些软件程序语言同样提供了内置宏以在应用中对字节顺序转换实现字节交换。

协议特定宏

所有通信协议必须定义协议的字节顺序，以使两端节点知道如何通信。像 TCP/IP 这样的协议，将网络字节顺序定义为大端模式，且 TCP/IP 包的 IP 头包含几个多字节域。小端模式体系结构的计算机在传输数据之前，必须将 TCP/IP 头信息中的字节重新排序为大端模式，类似地，需要将接收到的 TCP/IP 信息重排序为小端格式。

限制

在软件中实现交换功能会增加额外开支。位交换所引起的软件开支虽然存在，但是在需要处理的包的数量很多时开支引起的问题很容易修复，特别是在高频处理器中。

7.7 字节顺序中性代码

避免由字节顺序所引发问题的最好方式是在设计中使用字节顺序中性。可以通过两种途径完成这一任务：

- 将字节顺序选项作为软件可配置的选项。
- 在设计（IP）中使用字节使能，并将解码的任务留给系统或 SoC。

7.8 字节顺序中性编码指南

字节顺序中性代码可以通过标识外部软件接口实现，遵循下面的指南来访问这种接口[92]。

1）数据存储和共享存储体——数据必须以独立于字节顺序体系结构的格式保存。

可通过使用文本文件或指定仅用来保存数据的字节顺序格式（以使数

据总是写成同样的格式）等方法来完成这一任务。另一种更干净的方式是使用能理解所存储数据字节顺序格式的宏对数据访问进行包装，该宏与主处理器的理解方式一致。这种方式允许宏基于字节顺序进行字节交换。

2）字节交换宏——这与前面提到的几点没什么不同。宏（或包装器）能用于所有多字节数据接口进行交换操作。

3）数据传送——可以建立特定宏来从网络读数据或将数据写到网络中。根据输入数据的类型（如果它与本地主机字节顺序格式不匹配），可用宏来进行批量字节的交换。

4）数据类型——以本地数据类型访问数据。例如，总是以“int”类型来读/写“整数”，而不是以读/写4个字节的方式。一种可供选择的方法是使用自定义的字节顺序中性宏来访问多字节数字类型中的特定字节。不遵守这条指南会导致不同字节顺序体系结构间的代码兼容性问题。

5）位域——避免定义跨越字节边界的位域。

6）编译器指令——在使用会影响到数据存储（对齐、包装）的编译指令时要别小心。指令在不同编译器之间并不是总能移植的。“C”的定义指令（如#include 和#define）一般是没问题的。推荐使用#define 指令定义用于编译代码的编译器平台的字节顺序体系结构。

遵守字节顺序中性指南能使代码有更好的移植性，可以使同一源码运行在不同字节顺序结构的处理器上，从而减少了平台移植时的负担。

参考文献

1. opensourceproject.org.cn, Endian issues by GNU
2. Endianness White Paper by Intel, Nov 2004

第8章
消抖技术

8.1 简介

在电子设备内两个金属触点随着触点的断开闭合便产生了多个信号，这就是抖动。“消抖”是用以确保在每一次断开或闭合触点时只有一个信号起作用的硬件设备或软件。

机械开关和继电器触点通常由弹性金属制造，由传动装置强制接触。当触点撞击在一起时，它们的惯性和弹性共同作用形成抖动，导致产生一个快速的脉冲电流而不是从零电流到全电流的平稳转换。开关和线路中的寄生电感与电容进一步引起波形改变，形成一系列衰减正弦振荡。这个影响在交流电源线路中不明显，因为抖动太快不至于影响大多数设备。但在某些模拟和逻辑电路中可能产生问题，因为这些电路反应太快会将开关脉冲误当作数据流。

时序逻辑数字电路特别容易受到触点抖动的影响。开关抖动产生的电压波形干扰正常逻辑电路的振幅和时序规格要求。结果是亚稳定性、竞争、小脉冲和故障等问题导致电路失效。

当按下计算机键盘的一个键时，期待计算机记录下一个单触点。然而，事实上，第一次接触时，会有轻微抖动或接触接通，抖动结束时会有另一个接触，然后再产生抖动，如此类推。通常这些产品使用薄膜开关，尖端为导电的橡胶材料。按下去时与电路板上裸露的触点接触。由于橡胶是柔软的，因此软接触几乎不产生抖动。主要问题是大多数这类解决方案对高冲击力等不是很有效。

本章详述了消抖技术设计依据指南，目的在于实现平滑无抖动开关。

8.2 开关行为

图 8.1 展示了一个带有上拉电阻的简单按键开关。图 8.2 展示了当按键压下和释放时相应的输出。

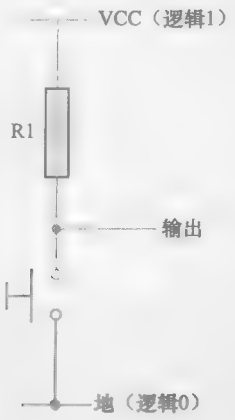


图 8.1 带有上拉电阻的按键开关

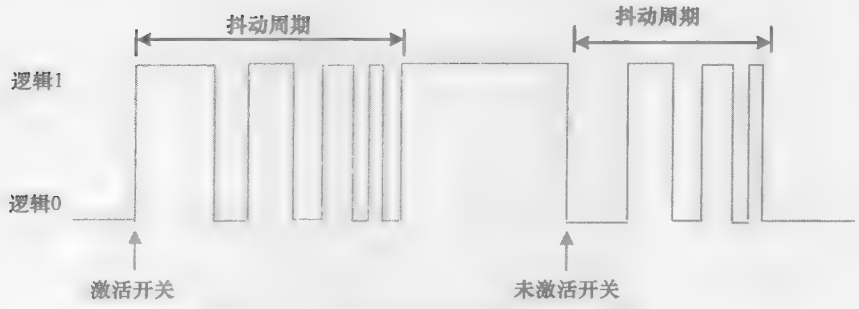


图 8.2 开关激活或不激活时的抖动周期

如果开关是用于开灯或启动风扇马达，则触点抖动不是什么问题。如果开关或继电器用作数字计时器、个人计算机或微处理器等设备的输入，由于触点抖动则会产生问题。计数器将会出现多次计数而不是一次计数。在开关释放时也会出现同样问题。

关注这个问题的原因是触点停止抖动所花的时间通常大约是毫秒级的，数字电路能在毫秒甚至更快时间（纳秒）内响应。

通常的解决方案是消抖设备或软件可以确保在一个给定时间内（通常为毫秒级）仅有唯一数字信号记录。在跳转到各种开关消抖解决方案前，我们先了解开关和抖动周期。

8.3 开关种类

最简单的开关类型是将两个电导体通过传动机构的运动联系在一起。其他开关更复杂，包括靠物理信号实现开关的电子线路（如光或磁场）。无论如何，任何开关的最终输出将是（至少是）一对接线端子，它们或通过开关内部触点装置连接在一起（闭合）或不连接（断开）。

图 8.3 显示了一些开关。

图 8.3 开关的种类

拨动开关是通过一个控制杆转到两个或多个位置中的一个位置来操纵的。家用电路常用的灯开关是拨动开关的一个例子。

按钮开关是双位设备，通过按钮的按下和释放来操纵。大多数按钮开关内部有一个弹簧机构，通过瞬间操作使按钮回到“出来”或“不按下”位置。

温度开关包含两个紧挨的金属薄片，每个金属片有不同的热膨胀率。当金属片加热或冷却时，两片金属间的不同热膨胀率使它们产生弯曲。这样，薄片的弯曲就可用于操纵开关触点装置。

压力开关使用气体压力或液体压力，压力通过活塞、薄膜或波纹管变为机械力来驱动开关。

液位开关也可设计用来探测固体材料的水平，如木材、粮食和煤炭等。

选择开关是由旋钮或某种控制杆来操纵到两个或多个位置中的一个位

置。类似于拨动开关，选择开关可能保持在任意的通断位置或内置弹簧装置回位。

可能有更多类型的开关，在此没有一一列举。但不同的开关运转方式不同，抖动周期也不同。简单廉价的开关比专用开关可能具有更大的抖动周期，例如，具有多个平行触点的开关抖动较小，但开关更复杂且费用高。有关开关设计的技术和指南数量众多，可用于减少抖动周期，但这不在本书范围内。

8.4 消抖

有几种办法可用于解决触点抖动问题（即输入信号“消抖”）。本节提出硬件和软件的解决办法。

8.4.1 RC 消抖

如图8.4所示，一个电阻－电容（RC）网路可能是最常见且最容易的消抖电路，就是把一个电阻和电容连接在一起，开关连接至中间。电容经过电阻充电，开关未使用的默认状态是高电平。当开关闭合时，它慢慢将电容消耗至地电位，以此减弱所有小抖动的影响。这种电路能承受某些抖动但不完全消除它们（见图8.5）。

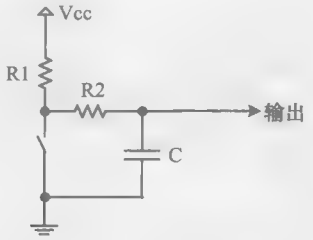


图8.4 RC 消抖电路

当开关断开时，电容器两端电压为0，但它以某个速率开始上升，这个速率由R和C的值确定。抖动接触使电压下降并降低电容的电荷聚积。为了完全消除抖动需要以R/C速率缓慢放电。R/C可调节至某个值，从而使得电压在抖动停止前一直低于门电路的逻辑1。这潜在一个副作用就是如果时间常数过大则开关对快速“断开”或“闭合”可能无法反应。

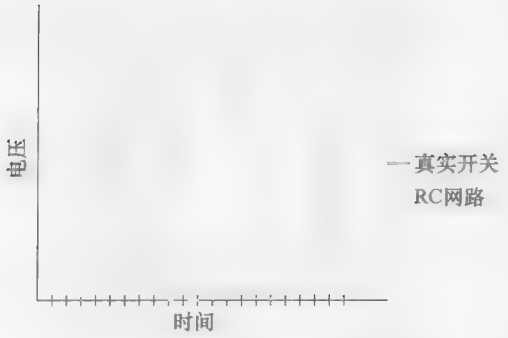


图 8.5 真实开关与 RC 网路对比

现在，假设开关暂时断开，电容器充满电。用户合上开关，电容器通过 R2 放电。电压又缓慢下降，一段时间内门电路输入保持逻辑 1。在抖动期间的一小段时间内，触点不断断开闭合。当触点断开时，时间即使很短，两个电阻就为电容再次充电，增强了门电路的逻辑“1”。选择合适的元件值以保证抖动停止前门电路逻辑保持为“1”。

以上所示 RC 电路即使没有 R2（R2 = 0），也能很好地消除任何抖动。快速操作开关会产生次微秒级或更小的抖动，因此具有更加快速的上升时间。更糟糕的是，取决于元件的物理排列，开关输入变为逻辑“0”时电容器两端的电压可能仍为逻辑“1”。当触点断开时，门电路为逻辑“1”。导致输出为一连串的“1”、“0”抖动。R2 使电容器慢速充电，忽略抖动的频率而产生干净的逻辑电平。电阻同样限制了流经开关触点的电流，防止电容器瞬间大量电涌烧坏触点。

最后，开关的状态信息实际上不是数字的，因此用它控制类似于开关集成电路的电路不会很有效。为了正确地使用开关状态信息，需要基本的模数转换，它由一个附加在 RC 网路上的逻辑门电路组成，如图 8.6 所示。

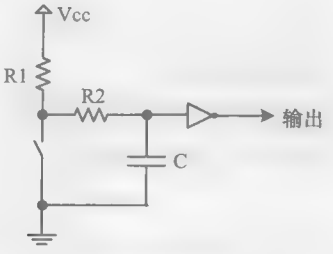


图 8.6 具有数字逻辑的 RC 网路

逻辑门电路有某个电压阈值，在此阈值要改变输出状态。它对开关抖动提供某些容差，但某些开关抖动可能会漏网，如图 8.7 所示。

图 8.7 RC 网路对比逻辑输出

逻辑门或反向器不会是一个标准的逻辑门。例如，TTL 逻辑电路定义逻辑“0”为 0.0V ~ 0.8V 的输入，逻辑“1”为高于 2.0V 的输入。介于 0.8V ~ 2.0V 的输出是不可预测的。通过使用带有施米特触发器的逻辑门电路可能增加对抖动的容差。有了施米特触发器，电压降至第一个阈值以触发，但电压上升到相同阈值，直到另一个更高的阈值前，状态不会改变。施米特触发器降低了对开关抖动的灵敏度，如图 8.8 所示。

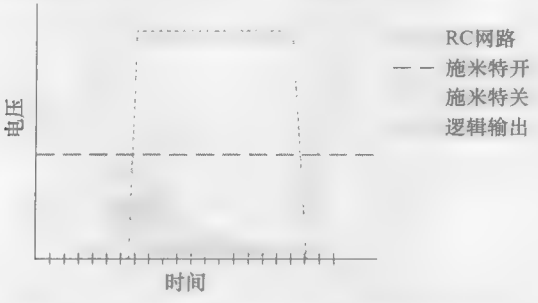


图 8.8 RC 网路对比逻辑输出（施米特触发器）

基于“施米特触发器”输入的电路具有滞后现象，输入抖动但输出保持已知的稳定状态。

试图对每个各种电路调节 RC 比例比较麻烦。我们提出对各种情况有效的通用 RC 电路。电容器放电定义如下：

$$V_{Cap} = V_{inital}\ (e^{-t/RC})$$

式中 V_{Cap} = 在时间 t 电容器两端的电压

$V_{initial}$ = 电容器两端的初始电压

t = 以秒为单位的时间

R = 以欧姆为单位的电阻值

C = 以法拉为单位的电容值

电阻 R 和电容 C 的值应当这样选择：V_{Cap} 直到开关停止抖动始终高于使门电路转换的阈值电压。

$R1 + R2$ 控制电容器充电时间，以开关为条件设置消抖周期。充电方程式为：

$$V_{threshold} = V_{final}\ (1 - e^{-t/RC})$$

式中 $V_{threshold}$ = 最坏情况下电容器两端转换点电压

V_{final} = 电容器两端最终充电值

图 8.9 展示了 RC 消抖电路的一个小变化，即在 $R1$ 和 $R2$ 间增加一个二极管。此二极管为一个可选组件以保持正确运行，即使出现 $R1 + R2$ 值小于 $R2$ 等错误导致滞后电压呈现不同值时也能正确运行。在此情况下，二极管形成一个捷径将 $R2$ 从充电线路中除掉。所有电荷从 R1 流过。

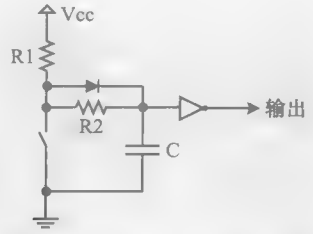


图 8.9 稳定的 RC 消抖电路

我们将更详细地分析这种情况。图 8.10 展示了开关分别处于断开和闭合时的电路状态。

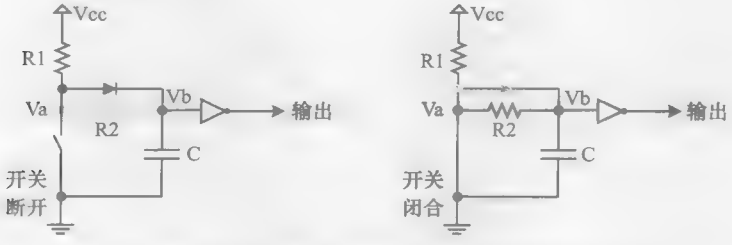


图 8.10 稳定的 RC 消抖电路状态（开关断开/闭合位置）

当开关断开时，电容器 C 取道 $R1$ 和二极管充电，最后，电容器充电，Vb 大约在 Vcc 以下 0.7V 范围内。因此反相施米特触发器的输出为逻辑“0”。

当开关闭合时，电容器取道 $R2$ 放电，最后电容器 C 放电，Vb 到达 0V。因此反相施密特触发器的输出为逻辑“1”。

如果抖动发生并有短时的开关断开或闭合，电容器将阻止 Vb 直接变成 Vcc 或地线 GND。尽管抖动会导致电容器的轻微充电和放电，但假设施密特触发器输入的滞后现象会阻止开关输出。

注意，要求电阻 $R2$ 作为电容器的放电路径，如果没有 $R2$，开关闭合时电容器会短路。在开关断开时，如果没有二极管，$R1$ 和 $R2$ 都将形成电容器充电路径。$R1$ 和 $R2$ 的组合将增加电容器的充电时间，降低电路的速度。另一个选择是将电阻 R1 容量变小，但在开关闭合而 $R1$ 连接在电源线两端时会导致不必要的电流浪费。

8.4.2 硬件消抖电路

图 8.11 展示了另一种硬件方法。它使用了由一对与非门电路制作的交叉耦合闩。它还可以用 SR 双稳态触发器设计。使用交叉耦合闩的好处是它提供一种完全的消抖而不用考虑延时限制，而且它的响应速度和触点断开和/或闭合一样快。注意，电路要求触点正常断开或闭合。在开关中，这种安排称为“双掷”。在继电器中，这称为“形状 C”。

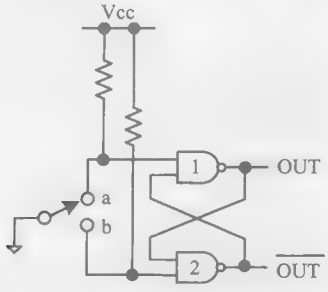


图 8.11 SR 消抖电路

当开关处于位置“a”时，门电路“1”的输出就是逻辑高（Logic HIGH），不考虑其他输入值。这使得门电路“2”拉到逻辑低（Logic LOW）。如果开关在触点间移动并不在这两点间的区域短暂停止，由于门

电路“2”将逻辑“0”回送到门“1”使其保持状态。这样保证了锁存器输出为非抖动。

接近以上思想的软件方法之一是将两个触点上拉输入至 CPU 的输入引脚，CPU 当然会注意到许多抖动，但通过编写探测两个触点之一行为的简单代码，抖动同样可消除。

8.4.3 软件消抖电路

软件开关消抖可以很简单，尽管算法的选择会依赖于应用及开关的操作。在转到开关消抖的软件技术之前，了解这个问题很重要。

检查开关的动态特征和评估环境的影响很重要。随着开关的断开或闭合，所有开关显示开关触点抖动。如前所述，在触点完全到达最后位置前，开关触点实际上要多次相互反弹。（如果开关位置对触摸敏感，人们可以不经意地触摸开关而引起抖动。开关制造商称这种不经意的触摸为“玩耍”开关）。环境干扰包括振动及最重要的电磁干扰（EMI）。

电磁干扰是一种有害的干扰，是由于外部来源发射的电磁辐射而影响电路。这种干扰会引起开关内噪声导致抖动。电磁干扰可通过适当的消抖程序来处理。

下面将描述一些软件（或固件）的开关消抖技术。

解决方案 A：在足以让抖动停止的时间后读出开关

开关消抖的一个简单解决方案是每 400 ~ 500 毫秒读出开关，并设置状态标志指示开关状态。查看开关性能，任何好的开关都会在这个时间内消除抖动，每 500 毫秒产生一个完全的输出。这种方法唯一不好的是响应时间慢，如果开关操作时间快于 500 毫秒这种方法会失败。但就绝大多数情况而言，这种方法还是有效的。

上述简单方法并不提供任何 EMI 保护。通过为开关提供足够的时间来停止抖动可以减少绝大多数随机噪声脉冲，但取开关状态期间单一短时脉冲波干扰可能被误当作触点转换。为避免这种情况，必须要修改软件在每 500 毫秒循环多次读取输入，以寻找一个稳定的信号。这样就可拒绝大多数 EMI。

解决方案 B：开关启动时中断 CPU 并在中断服务程序（ISR）中消抖

通常，在激活触点时，连接计算机的开关或继电器会产生中断。中断

将调用一个子程序（中断服务程序）。以下以一种通用汇编语言给出一个典型的消抖程序。

```
DR:     PUSH    PSW         ;   SAVE PROGRAM STATUS WORD
LOOP:   CALL    DELAY       ;   WAIT A FIXED TIME PERIOD
        IN      SWITCH      ;   READ SWITCH
        CMP     ACTIVE      ;   IS IT STILL ACTIVATED?
        JT      LOOP        ;   IF TRUE, JUMP BACK
        CALL    DELAY       ;
        POP     PSW         ;   RESTORE PROGRAM STATUS
        EI                  ;   RE-ENABLE INTERRUPTS
        RETI                ;   RETURN BACK TO MAIN
                                PROGRAM
```

设计思想是一旦激活开关，就调用消抖程序（De-bounce Routine，DR）。DR 调用另一个子程序——DELAY，其作用就是消耗足够长的时间使得触点停止抖动。此时 DR 检查触点是否仍在激活状态（也许用户把手放在开关上）。如果处于激活状态，DR 等待触点清零。如果触点清零，考虑到在结束前触点松开时的抖动，DR 再一次调用 DELAY 程序。

消抖程序必须调整以适用于你的应用。该要点不可能永远成立。程序员同样必须意识到开关和继电器随着使用年限增长将失去弹性。这会使触点停止抖动的时间增长。因此在键盘崭新时运行良好的消抖代码一两年后可能失灵了。请咨询开关制造商最坏情况下抖动时间的数据。

解决方案 C：使用计数器消除噪声并验证开关状态

另一个想法是制作一个计数器，在信号为逻辑低时计数，在信号为逻辑高时重置计数器。如果计数器到达某个固定值，该值应当比噪声脉冲大一至两倍，意味着电流脉冲为有效脉冲。

以下为 C 代码的一个样本快照：

```
// include files
unsigned char counter;  // Variable used to count
unsigned char T_valid;  // Variable used as the minimum
                        // duration of a valid pulse
void main(){
  P1 = 255;           // Initialize port 1 as input port
  T_valid = 100;      // Arbitrary number from 0 to 255 where
                      // the pulse if validated
  while(1){                   // infinite loop
    if (counter < 255){       // prevent the counter to roll
                              // back to 0
        counter++;
    }
```

```
        if (P1_0 == 1){
            counter = 0;      // reset the counter back to 0
        }
        if (counter > T_valid){
           //....
           // Code to be executed when a valid
           // pulse is detected.
           //....
        }
        //....
        // Rest of you program goes here.
        //....
    }
}
```

8.4.4 消抖指南

前一节讨论过各种消抖方法，然而浪费大量 CPU 周期来解决消抖问题并不是一个好主意。消抖是一个小问题，只能占用计算机一小部分资源。所以应当选择将 CPU 开销最小化的方法。以下给出了一些应当遵守的指南以在电路中拥有强大的消抖机制：

- 与消抖相关的 CPU 开销应当最小化。
- 非消抖开关必须连接至已编程序的 I/O 引脚，绝不能连接至 CPU 的中断。如果连接至 CPU 中断，抖动会引起多重中断，同样会增加 CPU 的负担，因为每次中断 CPU 都会执行中断服务程序（ISR）
- ISR 延时不能容忍，必须保持 ISR 的快速。与开关状态相关的中断不应当用作时钟或触发器的数据信号，因为这会违反了最小时钟宽度或数据建立及保持时间等原则。
- 开关输入取样的频率不应当与外界事件同步，否则会产生周期性的 EMI。以常用 50/60Hz 频率的速度取样应当避免。甚至机械振动也会产生周期性干扰。对汽车而言，甚至以发动机振动或驾驶杆振动同步的频率取样都可能引发 EMI。
- 系统应当对开关（用户）输入立即响应。万一开关状态显示在 LED 或显示器上，用户要系统迅速响应以避免显示器或 LED 出现混乱情况。
- 使用定时器有规律间隔地中断 CPU（例如，每几个毫秒），而不是用延时（以毫秒或秒为单位）来等待输入稳定。这样可使得消抖

代码可移植到新的编译器或 CPU 中，而不是随着每次时钟频率变化或 CPU 变化而改变等待状态。

8.4.5 在多重输入下消抖

在许多实际情况下，一个系统需要多组开关。单个输入开关消抖的方法对多重输入的单个消抖没有意义，所有输入开关可同时处理而占用 CPU 开销很少。本节将消抖技术或算法扩展至多个开关或输入。图 8.12 展示了具有多重输入开关的系统。

用于处理多重输入的消抖算法（伪代码）显示如下：

```
// This program demonstrates the simultaneous debouncing
// of multiple inputs. The input subroutine is easily
// adjusted to handle any number of inputs

Main:
GOSUB Debounce_Switches          // get debounced inputs
PAUSE 50                         // time between readings
GOTO Main                        // Continue the loop
END

Debounce_Switches:
switches = 0xF                   // enable all four inputs
FOR x = 1 TO 10
   switches = switches & ~Switch_Inputs // test inputs
   PAUSE 5                       // delay between tests
NEXT
RETURN
```

Debounce-Switches 子程序的目的是确保输入保持稳定 50 毫秒而没有触点抖动。消抖输入将以变量形式返回开关，在开关位置以一个逻辑“1”代表一个有效输入。

Debounce-Switches 程序开始时假设所有开关输入都是有效的，所有开关的位设为 1。然后，扫描输入并与 FOR-NEXT 循环中先前的状态比较。由于输入为低电平有效（按下时为 0），“1”的求补运算符将其反转。逻辑与（&）运算符用于校正当前的状态。一个开关要有效，它必须在整个 FOR-NEXT 循环中保持按下状态。

以下是消抖技术的工作原理：当开关按下时，开关的输入将为 0，如图 8.12 所示。“1”的求补运算符将“0”反转为“1”。1“与”1 还是 1，所以开关保持有效。如果开关没有按下，开关的输入则为“1”（由于 10kΩ 拉至 V_{dd}）。1 反转为 0，0“与”任何数数字还是 0，导致在整个循环过程中开关保持无效状态。

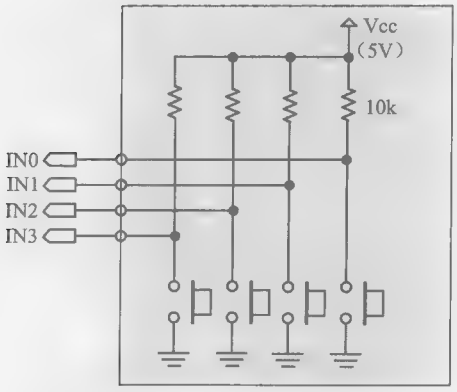


图 8.12 具有多个开关的电路

我们总是推荐通过时钟中断触发 Debounces-Switches 程序，而不是推荐消抖输入间 50 毫秒的固定延时，这样做是为了使设计具有可移植性。

8.5 现有的解决方案

对于不含外部输入的消抖电路设计而言，系统可选择使用外部消抖集成电路。在最常用集成电路中，MAXIM MAX6816/MAX6817/MAX6818 系列提供了单个、两个、八个开关消抖电路，它们为数字系统提供了机械开关的完全接口技术。如图 8.13 展示了 MAX6816 与任何需要消抖输入引脚但不包含内部消抖电路的微处理器或芯片的互联。

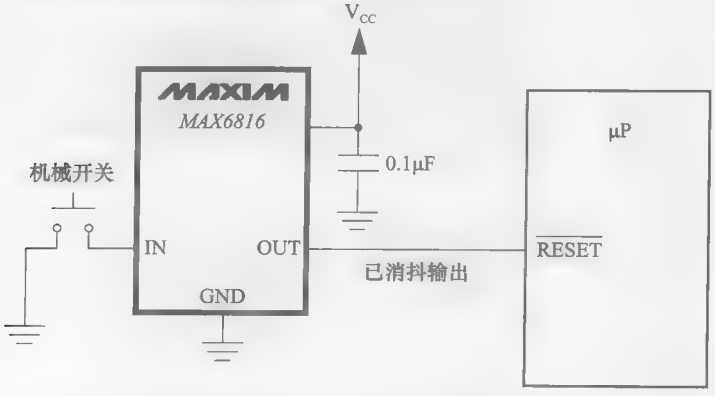


图 8.13 MAX6816 的消抖复位输入

MAX681x 系列产品接受一个或多个来自机械开关的抖动输入，并在短暂的预置限制延时后产生一个完全的数字输出。

MAX6818 八通道开关消抖电路用于数据总线接口。MAX6818 监控开关并提供开关状态变化输出（CH），简化了微处理器（μP）的定时询问和中断。

事实上，所有机械开关在断开和闭合时都会抖动。当开关断开或闭合时，消抖电路通过要求连续的时钟输入在若干采样周期内保持相同状态而达到消抖的目的。输入保持 40 毫秒的稳定状态输出才发生变化。

图 8.14 展示了包含芯片振荡器、计数器、“异或非”门、D 触发器的功能块。当输入不等于输出时，“异或非”门使重置计数器。当开关输入在整个限定周期内状态稳定，计数器为触发器计时，更新输出。

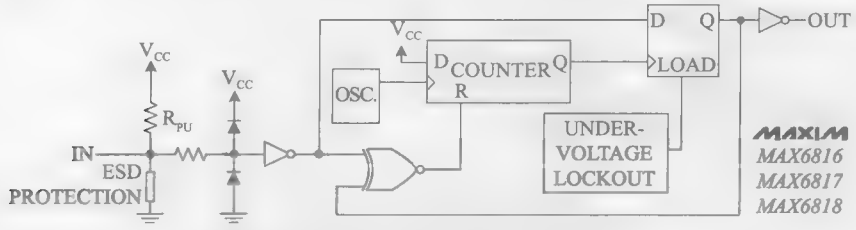


图 8.14 MAX6816/6817/6818 结构图

图片来源：Copyright © Maxim Integrated Products（http://maxim-ic.com），经授权使用

欠电压锁定电路确保在加压下输出处于正确的状态。当电源电压低于低电压阈值时，消抖电路保持通透。开关状态处于没有延时的逻辑输出。

除了消抖电路外，上述 Maxim 设备还包括在所有引脚上的 ±15kV 静电放电保护，以防止在处理和装配时遇到静电放电。

第9章

电磁兼容性能设计指南

9.1 简介

电子线路易于接收来自其他发射器的辐射信号，无论是有意或无意发射。这些电磁干扰（EMI）使得设备内毗邻的元件不能同时工作。这就有必要进行电磁兼容（Electro Magnetic Compliance，EMC）设计以避免系统内有害的电磁干扰。

以前，由于较差的EMC，发射器干扰电视图像正常显示是常见现象。现在随着电子装备现代化，我们可以几乎毫无干扰地在任何电子设备附近使用移动电话和其他无线设备。这都是由于确保了设备不产生多余的辐射，同时设备也不易受到射频辐射的干扰。采用好的EMC设计原则使这些成为可能。

注意

与大多数项目不同，EMC承诺不能只由设计来保证，必须经过测试。

9.2 定义

电磁兼容性是一个系统在预期的电磁环境中运行而不对其他系统产生不利影响或不受其他系统不利影响的能力[1]。

一个系统电磁兼容应满足：

- 不干扰其他系统；
- 不易受其他系统的干扰；
- 自身不干扰。

换言之，电磁耦合性包括辐射、免疫和自兼容。电磁兼容性的每一项包括三个因素：

a）源头，谁是噪声的发射体

b）受体，谁是噪声的接收者

c）耦合机理，通过这种机理噪声从源头传输到受体，并产生数种不同现象。

图9.1显示了4种基本的耦合机理：传导耦合、电容耦合、磁耦合或感应耦合，辐射耦合。每一种耦合途径可分解为一种或多种耦合机理的共同作用。

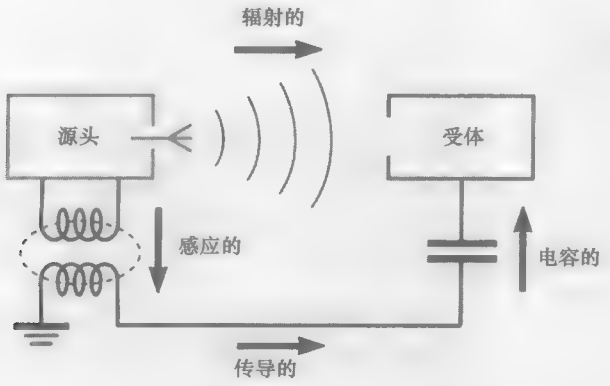


图9.1 4种耦合机理（图片源于：Wikipedia.org）

图9.1还显示，噪声源驱动电流，这种电流通过耦合途径（例如，PCB连接）流动并产生电压降。这种电压扰动通过耦合途径传输给受体，如果这个电平足够高则导致其功能失常。因此，采用好的设计原则避免这种情况非常重要。

下面介绍本章将要使用的一些术语。

一个很相关的术语是电磁干扰（EMI），这是一个电子设备中破坏性的电磁能量通过辐射或（和）传导途径传输到另一个电子设备的过程。

一个设备的电磁敏感度（Electro Magnetic Susceptibility，EMS）水平是指对电气干扰和传导式电气噪声的抵抗能力。静电放电（Electrostatic Discharge，ESD）和快速瞬间触发（Fast Transient Burst，FTB）试验确定一个设备在不良电磁环境中运行时的可靠性水平。

第三部分是图 9.2 显示的从源头到受体的非预期途径。

这样，如果电动剃须刀厂商遵循了必要的电磁兼容性设计原则，则在听调幅广播时使用电动剃须刀也没有什么问题了。在此例中，电动剃须刀电动机电刷的电弧放电就是一种意外辐射；调幅广播通过相关途径（电线和/或通过空气）接收到噪声就是不必要的敏感度。

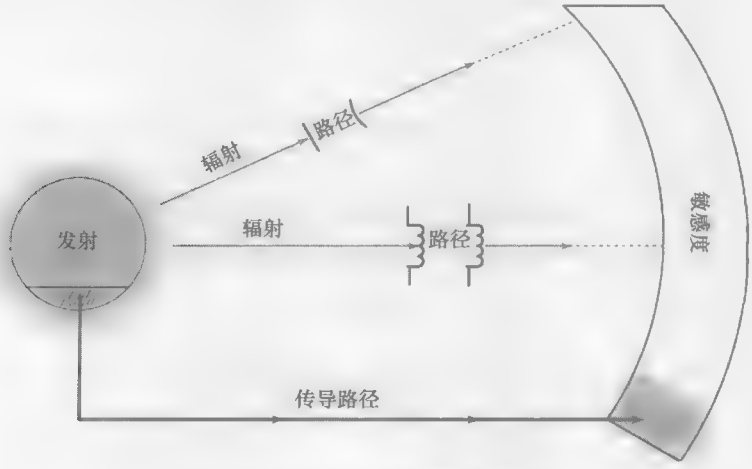


图 9.2 描述 EMC 范例的框图

9.3 电磁干扰理论及与电流和频率之关系

电流是产生电磁辐射的关键。当微控制器速度增加，电流的需求也增加。电流流过一个回路则产生磁场，磁场大小与回路面积成正比。回路面积定义为线路长度乘以到接地板（the ground plane）的距离。当信号改变逻辑状态时，变化的电压产生电场。这样，电流回路生成了辐射。下面的方程式显示了电流、回路面积与电磁干扰频率的关系[3]。

$$\mathrm{EMI}(V/\mathrm{m}) = kIAf^2$$

式中：k = 比例常数

I = 电流（A）

A = 回路面积（m^2）

f = 频率（MHz）

由于线路板层叠要求，到接地板的距离通常固定，因此，减少辐射的关键就是减少电路板设计的线路长度。

9.4 电磁干扰的规程、标准和认证

目前存在几种有关 EMS 和 EMI 问题的标准，每种标准都有其应用领域，用于制成产品或设备。到目前为止，还没有应用于子体系或电子元器件的官方标准。然而，EMC 测试必须在子系统中执行，以用于评估和优化电磁兼容性性能的应用。

表 9.1 和表 9.2 显示了常见的 EMC/EMI 标准。

额定功率小于600 瓦的个人计算机、显示器和电视机等，必须符合 EN 61000-3-2 D 类谐波限制。照明设备必须符合 C 类谐波限制。便携的工具和非专业的弧光电焊设备必须符合 B 类谐波限制。所有产品必须符合 A 类（见表 9.1）谐波限制。

表 9.1 电磁发射

标 准	相等的国际标准	描 述
EN 50081—1		通用发射标准——住宅
EN 50081—2		通用发射标准——工业
EN 55011	CISPR① 11	用于工业、科学和医学（ISM）的无线电频率设备
EN 55013	CISPR 13	用于广播收音机及相关设备
EN 55014	CISPR 14	用于家用电器、电动工具
EN 55022	CISPR 22	用于信息技术设备，如计算机等
	FCC②, Part 15	无线电频率设备——无意的辐射体
	FCC, Part 18	工业、科学和医学设备

①CISPR：国际无线电干扰特别委员会
②FCC：美国联邦通信委员会

表 9.2 电磁敏感性

标 准	相等的国际标准	描 述
EN50082—1		一般抗扰标准——住宅
EN50082—2		一般抗扰标准——工业
EN50140	IEC 61000-4-3	辐射、射频、电磁场抗干扰性试验
EN50141	IEC 61000-4-6	对射频场感应引发的传导干扰的抗干扰性试验
EN50142	IEC6 1000-4-5	浪涌抗干扰性试验
	IEC 61000-4-4	EFT/触发抗干扰试验
	IEC 61000-3-2	谐波电流辐射的限制

9.5 影响集成电路抗干扰性能的几个因素

当今低成本微控制器的半导体工艺技术可以实现晶体管栅极长度在0.65～0.090微米之间，这些栅极长度能产生和响应生成时间为亚纳秒级的信号。因此，微控制器能响应注入其引脚的静电放电信号或EFT信号。除制造工艺外时，微控制器对于ESD或EFT事件的性能还取决于集成电路设计和封装、印制电路板（PCB）的设计、微控制器上运行的软件、系统设计和到达微控制器时ESD或EFT波形的特征。

在过去，当对EMC一无所知时，通常的做法就是，对一个未考虑过任何EMC的现有产品，加装必要的滤波器、电涌保护器，采取屏蔽等各种手段使它满足EMC。这是最不合理的方法，因为这样做的成本过高而结果不如预期。

当设计新产品时，必须从开始阶段就着手考虑遵循EMC指南，这对低成本的解决方案尤为重要。好的PCB设计在生产中并不比差的成本高，但修理差的PCB费用就高了。一个设计者所犯的最昂贵的错误就是认为EMC放在最后处理。

对于小批量快速上市的系统，如果不是大批量低成本应用，使用贵的元件仍是合理的，更好的方法是在设计上投入更多时间和资源以减少最终产品的总成本。

下面推荐了更好的EMC设计指南。在此之前，我们先看看一些影响EMC的因素。

9.5.1 作为噪声源的微控制器

在微控制器的应用环境中，静电放电、电源、高电流或电压下的开关、射频（RF）发生器等就是产生电磁干扰或噪声的一些因素。

微控制器（或其分支电路）既是源头也是受体。

- *电源和地线中的电流*：作为CMOS设计的一部分，时变电流在电源和地线中流动，以数种不同方式产生辐射和传导EMI。
- *振荡器行为*：当振荡器为微控制器提供时钟脉冲源时，它就是一个连续的RF源。在振荡器电路各部分流动的任何时变电流都是重要的发射源，包括流经输入、输出、电源和接地部分的电流。

- 系统时钟电路：系统时钟可能是系统中最大的噪声源，尤其随着当今 PC 和工作站需要更高的时钟频率。这种辐射主要由基本谐波和低阶高次谐波产生，遗憾的是，常与收音机的常用高频 FM 波段冲突并干扰。这就迫使监管机构对可能使用时钟并产生辐射的 PC 和电子设备施加电磁辐射限制。
- 输出行为：任何常见单片机的输出行为包括时钟输出、数据和地址信号，都是潜在的发射源。在内部开关中的电流也同样涉及开关工作载荷，但负载电流流过的途径更长。EMI 源于时变负载电流，这些电流不仅在信号线路中流动，还流到地线或电源线中。输出信号相对权重取决于变换的频率和持续时间，即，转换时间越短，频谱越丰富。除此之外，输出线路的信号会产生串扰、开关噪声和反射。
- 开关噪声：开关噪声是指当一个信号激发起通路电感和负载电容的谐振而产生的干扰信号。开关噪声大到引起误操作才会引起关注，而它同样会增加额外的谐波量从而增加 EMI。
- I/O 开关：I/O 开关的负载包括封装引脚和焊线电感。最坏情况下的噪声依赖于开关时间。

对于一些微控制器，存储空间（地址/数据总线）通常是外部的（如 SRAM、DDR 等），这意味着信号在数个线路上的连续转换，这将对总体的电磁兼容产生重大影响。

9.5.2 影响电磁兼容性的其他因素

除微控制器外，其他影响电磁兼容性的因素包括以下几个。

- 电压：较高的电源电压意味着更大的电压摆动和更多的辐射。较低的电源电压影响敏感性。
- 频率：频率越高发射越多。当晶体管因开关通断时，高频数字系统则产生电流尖脉冲，增加了总体的噪声。
- 接地：无论是发射、敏感性或自兼容性等所有电磁兼容性问题，最重要的问题是接地不充分影响 EMC。单点接地在低频率下是可以的，但高频率下有高阻抗性就不行了。多点接地最适用于高频率应用，如数字电路等（见图 9.3）。

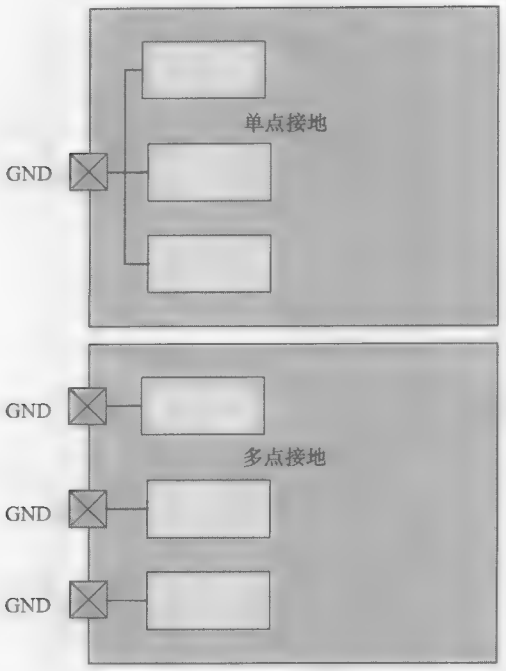


图 9.3 接地方案

- 集成电路设计/印制电路板：芯片尺寸、制造工艺、版面布局（多点接地和电源引脚改进）和封装方法都会影响 EMI。而且，恰当的印制电路板布局对防范电磁干扰也必不可少。

注意

9.5.2 节解释过的一些因素（如电压和频率）也适用于微控制器。

9.5.3 噪声载体

EMI 可通过电磁波、传导、容性/感性耦合来传递。EMI 必须到达导体才能干扰元器件。这就意味着导体的回路、超长或大面积容易受到 EMI 影响。

9.6 减少 EMC/EMI 的技术

防止干扰的三种办法是：

1）在源头抑制发射。

2）耦合路径尽可能低效。

3）受体几乎不受发射影响。

本节提出了处于不同抽象层次的常用噪声降低技术。提出的技术不是EMI的完整解决方案，但是采用这些技术可以极大影响存在噪声的系统的性能。

9.6.1 系统级技术

9.6.1.1 展频时钟技术

在数字系统内，周期性的时钟信号是EMI辐射的主要原因。此外，控制与计时信号、地址和数据总线、互连电缆和连接器都会产生EMI发射。

屏蔽是通过覆盖发射位置来减少EMI发射的一种简单方式，但额外增加了重量、空间和费用。通常屏蔽使得自动化生产变得困难，因而大量增加了人力劳动。

用低通滤波器减少EMI也有自身层面的问题，如对高速系统无效，因为这种滤波器既减少了关键的建立时间和保持时间的裕量，同时增加信号过冲、下冲和振荡。除此之外，滤波器的另一个主要问题是技术不是系统性的，即在任何一个指定节点降低EMI发射并不能减少在其他节点的发射。

一个更为有效的方法是使用展频时钟技术（Spread Spectrum Clocking，SSC）来控制和减少电磁干扰辐射。展频时钟发生器通过将辐射传播到更宽频带的方式来减少辐射发射（如图9.4所示）。通过在几百千赫兹频率

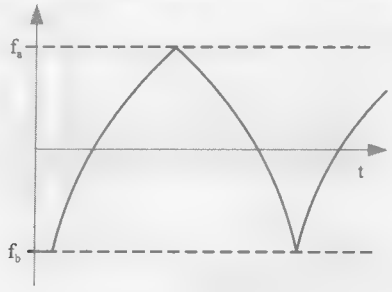


图9.4 典型的调制曲线

范围内调制处理器时钟，这个频带可以随着测量的辐射量减少而增宽。这样，SSC 会按预定的途径（即，调制曲线）用预定的调制频率调制时钟频率/周期，而不是维持恒定的系统时钟频率。

调制频率通常选择为大于 30kHz（高于声频带），常用的是用 30 ~ 90kHz，这样从源头上来控制和减少 EMI 发射。调制频率的上限值选得足够小，以避免系统的定时问题和跟踪问题。

比较其他减少 EMI 的技术，SSC 系统性的性质有一个很大的优势，因为所有时种和来自 SSC 时钟的时序信号都是以相同百分比来调制的，导致整个系统内 EMI 极大地减少了。

SSC 产生带有边带谐波的频谱。有意地扩宽系统重复时钟窄的频带，同时减少系统时钟基频和谐波的谱密度峰值。

除了减少 EMI 外，SSC 还有利于电路板线路与时钟驱动器驱动负载之间的阻抗匹配，这是时钟信号完整性的一个重要考虑。

9.6.1.2 差分时钟

差分时钟要求时钟发生器同时提供时钟和反向时钟，反向时钟有与主时钟大小相等方向相反的电流，相位差为 180°。必须注意电路板设计者需要确保主时钟线和反向时钟线按已选择路线并排在一起，这一点非常重要。时钟信号在负载端通过差分放大器接收，这意味着合格的时钟波形是两条时钟线上信号的差异。

差分时钟引起 EMI 减少是由于磁场对消的原因（见图 9.5）。由于磁场按右手法则随电流流动，而相位差为 180°的两股反相电流所产生的磁场相互抵消。减少磁场导致辐射降低[3]。

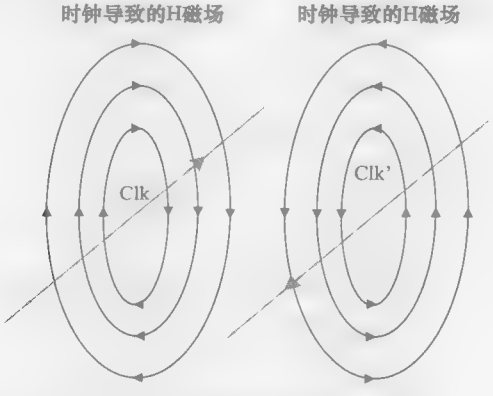


图 9.5 差分时钟中的磁场对消

与单端时钟不同，单端时钟产生的噪声出现在基准面，并可能耦合到输入/输出线中。与此不同，差分时钟回路是反向时钟信号，比基准面提供了更好的隔离，减少输入/输出线耦合，从而减少 EMI。

通常两条线（时钟线和反向时钟线）应尽量靠近。将地线放在差分时钟线外侧可以进一步减少辐射。

9.6.2 板级技术

本节仅包含芯片设计者必须知道的基本板级技术，以便让设计者对在芯片上执行的逻辑与在电路板上执行的逻辑间进行权衡。超出本节的详细内容则不在本书的范围之内。

9.6.2.1 电源输入滤波

消除动态干扰问题的首要机会在应用电路的电源或信号输入点。如果能够在此处充分抑制干扰信号，就可能不需要其余的软硬件技术了。除了能降低材料成本，还能降低或者消除不合规引起的风险。

图 9.6 显示，在输入点没有加装滤波器的情况，传播到电路板 1 的传导干扰信号能够辐射或耦合到电路板 2。

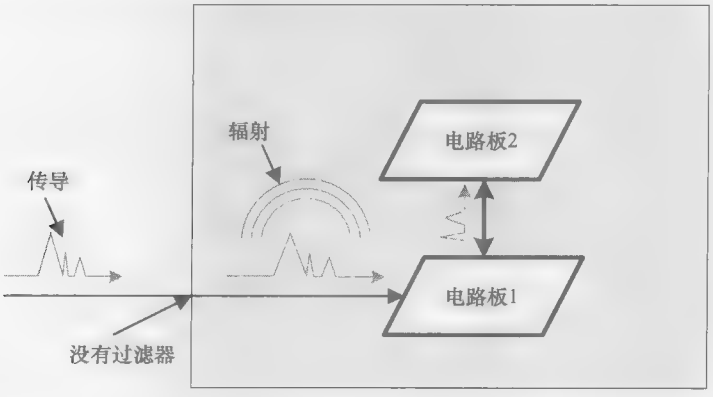


图 9.6 输入点没有加装滤波器

图 9.7 所示，在输入点加装滤波器，有助于抑制传导的干扰信号，从而向电路板 1 提供干净的信号而不会产生内部辐射。

在电源和信号连接到应用电路之前，在输入点瞬态抑制应进行优化，否则由于干扰信号失控，兼容性问题将复杂化。结果导致需要后续的软硬件技术以确保好的 EMC 性能。

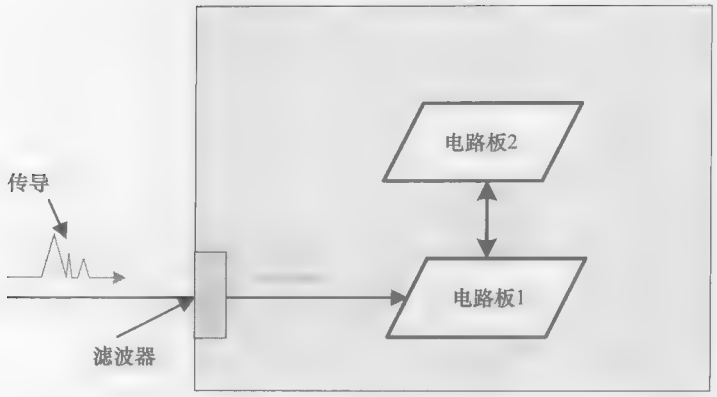


图 9.7 输入点加装滤波器

适入输入点应用电路的瞬态抑制器件随时可从众多供应商购买，如有需要，还可以设计定制方案。

9.6.2.2 更多的滤波器

当信号噪声源不能消除时，推荐把滤波器作为最后的手段。EMI 滤波器和铁氧化物磁珠是常用的滤波器。铁氧化物磁珠可以增强电感以抑制高频。

电磁干扰（EMI）滤波器

在商业中 EMI 滤波器用于消除电源线中的高频噪声。EMI 滤波器通常由电容器与电感器组成。需要 EMI 滤泔器的节点阻抗决定了电容器和电感器的配置。高阻抗节点需要电容器，低阻抗节点需要电感器。

EMI 滤波器也可以由穿心电容器、L 型电路、PI 型电路和 T 型电路等组成。穿心电容器的主要部件就是一个电容器。当连接到滤波器的阻抗是高阻时穿心电容器是不错的选择。图 9.8 描述了穿心电容器。穿心电容器并不提供节点间高频电流隔离。

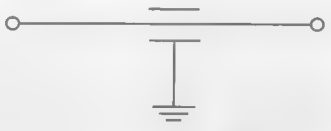


图 9.8 穿心电容器

图 9.9 所示为 L 型电路，在电容器旁有一个电感器。这种配置最适用于输人和输出阻抗差别很大的情况。感性元件连接至最低阻抗端。

图9.9 L型电路

图9.10所示为PI型电路，两个电容器围着一个电感器。当输入和输出阻抗差别很大时，PI型电路最适用。当需要高阶衰减时也可用PI型电路。

图9.10 PI型电路

图9.11所示为T型电路，一个电容器两侧均有一个电感器。当输入和输出阻抗都是低阻时最适用。

图9.11 T型电路

其他可选方案是在电源输入点使用铁氧体磁环，这对于衰减1MHz以上频率又不导致低频功率损耗是经济便捷的方法。它们体积小，一般可以套在元件引线或导线上。

9.6.2.3 元件布局

子系统、部件和连线的布局很重要。有噪声的子系统、部件、连线应当与敏感的电子器件（如微控制器）物理隔离，以使辐射噪声耦合最小化。物理隔离可采用距离分离或屏蔽的形式。建议遵循以下指导原则。

- 将供电电路与模拟和数字逻辑电路分开。最简单的办法是采用独立的PCB来放置供电电路。
- 将与同一时钟线相关的所有部件尽量靠近放置。这可减少走线长

度，进而减少辐射。

- 将大电流器件尽可能靠近电源。
- 电路板高频部分尽量不使用插座。插座会引起高电感和阻抗失配。
- 将晶振、振荡器和时钟发生器远离输入/输出端和电路板边缘。这些器件产生的 EMI 会耦合到输入/输出端口。
- 将晶振平放在 PCB 上，从而减少到地的距离，并在电路板上产生更好的电磁场耦合。
- 将晶振的固定带接地。如果不接地，这些固定带会像天线一样产生辐射。

9.6.2.4 接地路径

许多 EMC 设计技术的基本思想是控制所有信号的接地路径，确保该路径远离可能受干扰的信号和电路。对于发送的噪声，这意味着在它离开系统前确保噪声找到接地路径。对于接收的噪声，这意味着在它到达系统敏感部分前确保找到接地路径。

除了 9.5.2 节提及的有关接地需要考虑的事项外，接地布局是至关重要的。下面给出一些好的接地布局建议。

- 不要将接地层和电源层分开。
- 为减少接地噪声耦合，将数字地与模拟信号地分开以减少耦合。
- 不要改变有信号走线的层，这可能导致增加回路面积而增加辐射。
- 连接所有接地通孔到每个接地层，同样地在等电位下连接每个电源通孔到所有电源层。
- 使接地层至少比电源层长 5 倍于两层的距[3]，这使得交流电的电位差能被接地层吸收。

9.6.2.5 线路布线

传输高速信号的线路必须十分小心地布线。平行走线哪怕距离很短也会产生容性和感应串扰。

在电容耦合中，源的上升沿会引起受体的上升沿。试验电路板特别容易出现这些问题，因为每排中过长的金属片在线路之间产生几皮法的电容。

在电感耦合中，受体中电压的变化与源的沿变化方向相反。电感耦合是电磁干扰的一种形式，电线或线路中电流的变化通过电磁感应导致另一个电线线路产生电压。

大多数串扰例子都是容性串扰。受体中噪声的大小与平行走线的距离、频率、发射源中电压波动幅度、受体的阻抗等成正比，与间距成反比。

电感耦合常见于低频率能量源，而高频率能量源常用电容耦合。

根据美国联邦通信委员会的限制，当线路长度超过波长的1/10时通常就变得重要了。军用标准的规定是波长的1/20至1/30。对于汽车和消费电子的两层板，达到波长的1/50就开始变得严重了，特别是在没有屏蔽的应用时。在超过以上这些范围时，线路就像天线一样，会增加辐射。线路超过4英寸就会产生FM波段噪声。在这种情况下，建议采用一些形式的终端负载来防止振荡。

以下的一些布线指南会有助于防止辐射或串扰。

- 与微控制器连接的传输射频的线路应当远离其他信号以免接收到噪声。
- 为增加线路间的隔离读，相邻线路间的距离应当增大，或者在关键线路的两侧增加保护线路。在相邻的布线层中间增加屏蔽层也是个好办法。
- 不要在晶体、振荡器和时钟发生器下布线，因为这些电路容易接收到噪声。
- 为了控制电路板边缘的线路周围的电磁场，应使线路离板沿距离大于线路距接地层的高度[3]。这样可使得线路周围的电磁场更易于和接地层耦合，而不会和附近电线或其他电路板耦合。
- 当线路从一层转换至另一层时，如果两层电路板与电源层/接地层不等距，则必须改变线路宽度和间隔以保持线路阻抗。如果在回路区域进行各种可能处理的结果都会增加辐射，则应尽量避免换层。
- 可能受到噪声干扰的信号在其下面应当有返回地，用于减小其阻抗，因而减小噪声电压和辐射区域。
- 如果有可能，则把若干会产生噪声的线路布在一起，周围布一些接地线。

9.6.2.6 制作分区

处理EMC问题一个较好办法是将PCB/电路板分隔成更小的区域，在每个区域中处理问题。这包括在电路板上界定元件的大概位置以便将辐射最小化。分区通常是同一块电路板上的不同区域。

图 9.12 展示了印刷电路板系统划分为不同分区的例子，分区 1 包括关键部分，而分区 2 和分区 3 包括非关键部分。

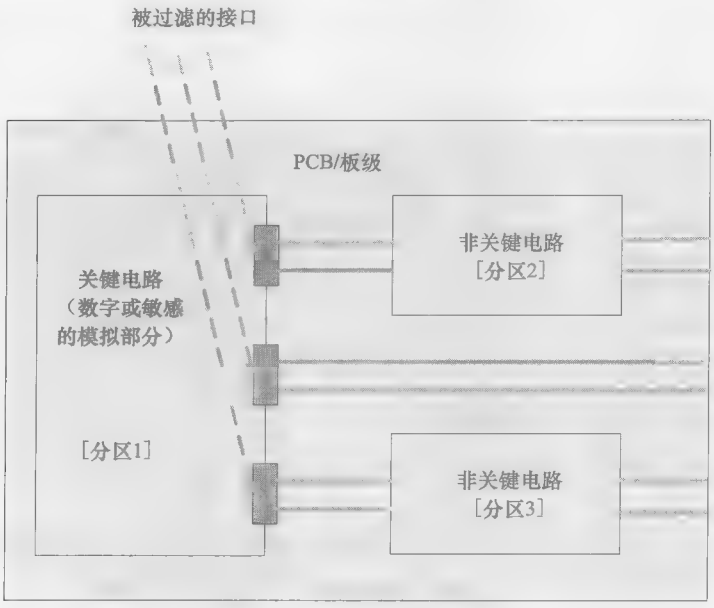


图 9.12 电路板分区方案

进出一个分区的所有线路需要几种滤波器。每个分区的设计者应当清楚一个分区会发出什么样的噪声和能忍耐什么样的噪声，还应考虑分区对分区的噪声辐射。

分区有很多种方法，图 9.12 给出了一种特殊的例子，在这个例子中，电源、数字电路和模拟电路放在不同的分区中，从而将噪声电路和敏感的电路分开。

图 9.13 是将一个分区放在另一个分区内部的另一种方法。

从最里面的分区进出的噪声将通过多层滤波器以对噪声产生更好抗干扰性。对于一个典型的基于微控制器的系统而言，最内层分区可能包含噪声最大的信号，如带有内存和其他高速接口的微控制器。离开这个分区（分区 1）的所有传输线必须经过过滤，以确保没有最高频的噪声传出去。下一级过滤可以在数字或模拟分区（分区 2），可能的第三级过滤可以在系统输入/输出端口上（分区 3），以进一步减少发出的噪声，如图 9.13 所示。

下面给出分区时可供考虑的一些更多指南。

- 微控制器等高速逻辑电路应当靠近电源放置，低速部件可以放远

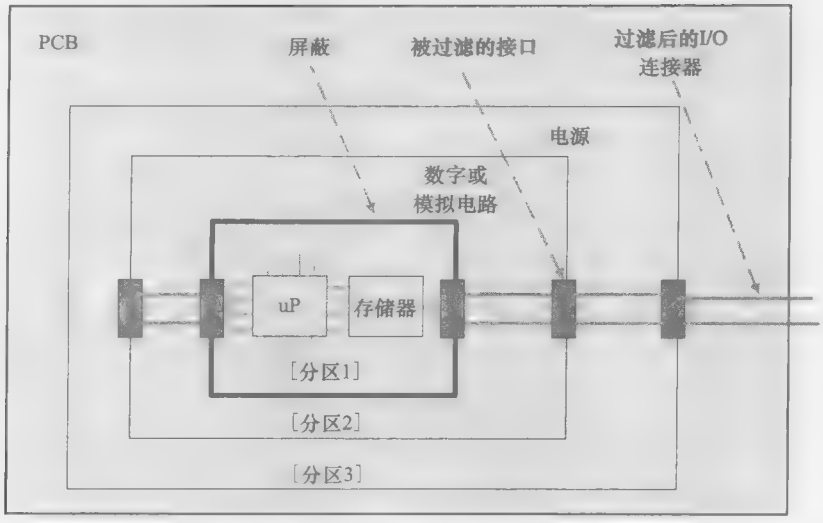


图 9.13 电路板分区的另一种方法

一些，而模拟部件可以放得更远。这样的安排使得高速逻辑部件几乎没有机会干扰其他信号线路。

- 振荡器应当远离模拟电路、低速信号和连接器。
- 微控制器应紧靠稳压器，而稳压器紧靠进入电路板的“电池电压”。

确定分区划分必须考虑的另一个因素是材料成本，因为这项技术的费用可能很高。

9.6.2.7 电源耦合

当逻辑门发生翻转时，供电线路就会产生瞬态电流。这些瞬变电流必须衰减和过滤掉。

$\frac{di}{dt}$高的电源产生的瞬态电流会引起地线和信号线产生“抖动”电压。高的$\frac{di}{dt}$会产生宽频谱范围的高频电流，从而激励结构及连线产生辐射。流经固定电感（L）的导体的电流变化导致的电压降为：

$$V = L\frac{di}{dt}$$

通过减小电感或者减少电流变化可以减少电压降。

低电感的高频陶瓷电容器最为理想。选择电容器要考虑的重要特性是

最大额定直流电压、寄生电感、寄生电阻和过电压失效机理。当在超过最大额定电压的条件下使用时，电容器应当选可自动修复的类型，如金属化聚酯薄膜电容器。

必须注意电容器不适用于分流由闪电、浪涌或切换大电感负载产生的较大的瞬变电流。

去耦电容用于以下两个目的。

- 对于吸收或者提供高频电流的器件来说，它们是充电源。如上所述，去耦电容减少电压骤降和接地漂移。
- 电容器为电源层的高频返回电流提供了一种接地途径。如果不用电容器，则电流通过输入/输出信号或电源连接器返回接地，从而产生大的回路，增加辐射。

旁路电容器在特定频率下会产生自谐振，这种现象必须考虑。

$$f=\frac{1}{2\pi\sqrt{LC}}$$

这样，有 3nH 电感的 200pF 的电容器将在大约 205MHz 谐振。

对于高于自谐振频率的噪声信号，旁路电容器变为感性而无法过滤这些信号。

最好是为每个元件都配置旁路电容器。然而，如果无法为每个有效元件提供旁路，就可以将精力集中在高频器件而忽略频率较低的器件。

9.6.2.8 印刷电路板配电和去耦电容器

配电系统的设计对确保印刷电路板 EMC 至关重要，这是 EMC 控制的基础。

接地和电源网路应当设计为平面或短宽线路。避免使用通孔和跳线来连接不同的接地区域。通孔和跳线将增加电感，在电路间产生共模阻抗噪声，从而导致功能退化。

微控制器有宽的电压范围，并且在时钟脉冲边缘的极短尖峰会从电源吸取电流。当 I/O 线紧拴在一起时，尖峰可能会有较大的幅值。随着绑在一起的 I/O 线的数量变化，电源的电流脉冲变化范围可能会很大。必须用去耦电容将这种电流尖脉冲在长电源线中释放掉（同样如前所述）。当规划接地和配电系统时，去耦电容的位置（或其他相关滤波器）也非常重要。

图 9.14 给出了一个去耦不充分的例子。电容器离微控制器过远以致产生大的电流回路。结果噪声很容易扩散到电路板上的其他器件。除了作为大电流的回路外，接地层也充当了这种噪声的天线。

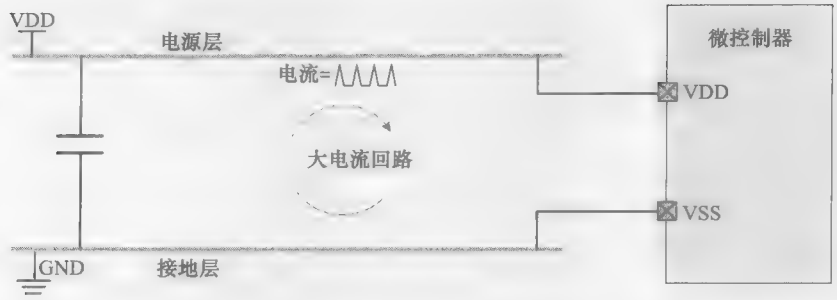


图 9.14 错误的去耦合：电容器离微控制器太远

图 9.15 给出的是电容器放在离微控制器较近位置的较好例子。作为大电流回路的线路并不是电源层或接地层的部分，这样避免了任何噪声的广泛传播。

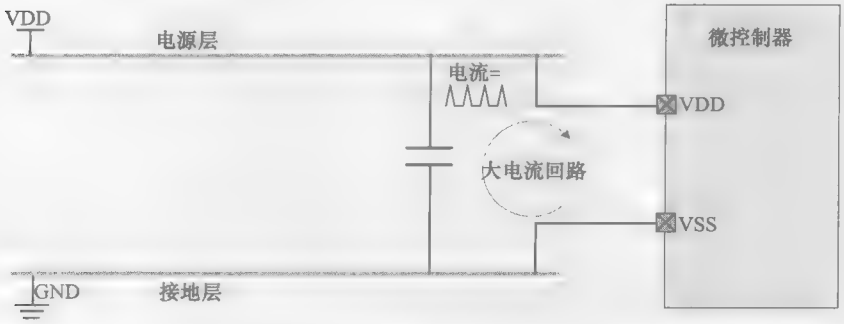


图 9.15 去耦电容器靠近微控制器

图 9.16 给出了另一个改进的例子，通过增加一个串联电感器减少电源层的开关噪声。选择的电感器数值应当使电压降可忽略不计。

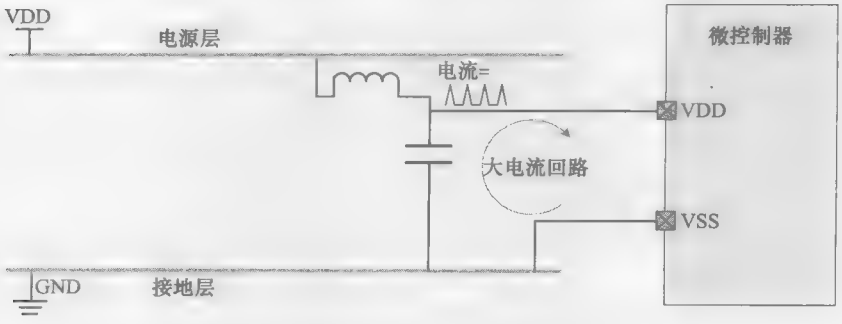


图 9.16 使用去耦电容器和串联电感器的情况

为了更好和更有效地去耦，建议将电源线和接地线（或引脚）紧挨放置。作为 EMC 的关键设计，应当尽可能多地设计电源/接地线对将电流分为多条路径。电源到地的电流分流到许多更小的回路中，从而明显改善 EMC 性能。

9.6.3 微控制器级技术

解决噪声问题的最佳途径是在源头。在大多数情况下，面向 EMC 的微控制器增加了应用的安全性和可靠性，实现成本低，可节约材料成本。此部分提供了提高 EMC 性能的微控制器级技术。

9.6.3.1 多时钟和接地

如前面几节所述，多个电源和接地引脚有助于将大电流分流到多条路径，以避免在静电放电和闩锁事件期间损坏有源逻辑和电路。同样，较小的峰值电流使得选择外接电源 - 地去耦电容器变得容易。

选择去耦电容器必须依照以下条件。

a）电容器的容量应足够大以在转换时间内提供所需的电流。

b）电容器的容量应足够小以使时钟频率小于电容器的谐振频率（如前所述）。

较小的峰值电流减少了两条规则产生冲突的机会。例外是芯片上有足够数量的去耦电容。在这种情况下，地（VSS）应当比电源（VDD）分配更多引脚，而电源总线应当仔细地相互连接以减少去耦电容器与电路之间的阻抗。

此外，还应遵守以下指南。

- 在所有电源/地线对中尽可能是均衡电流。
- 除了 ESD 保护需要外，应避免在内部连接电源引脚和接地引脚。
- 在芯片上使用独立的电源 - 地线对来把有噪声的电路与敏感电路隔离开来（除了基板设计指南特别说明）。

9.6.3.2 消除竞态条件

竞态条件定义了一种条件，即，器件的输出取决于输入端几乎同时的两个或多个事件，导致设备输出的转换。这就额外增加了系统噪声，在好的 EMC 设计中必须避免。

9.6.3.3 降低系统速度

提高 EMC 的一项关键参数是降低系统工作频率至绝对最小值。这包括主时钟、派生时钟和内部接口。

相对于符合所有性能需求的足够好的系统频率，仍建议对中断、CPU 处理时间等实时事件以及一个时间窗口内数据采集等任意事件顺列进行详细分析，以达成正好符合性能需求的最小时钟频率。在完成设计前可以使用建模工具来模拟系统以提供性能数据。

9.6.3.4 驱动器规格

当驱动器对于负载充电的速度快于负载所需的速度时，过快的沿速度会将导致过冲或下冲。快的转换速率会导致噪声以信号反射、串扰和接弹反弹等形式产生。

不要试图使用最快的转换速率和最大的驱动电流。通常，厂商提供指南会说明在给定电流激励下可同时转换的最大输出数量。所以减少 EMI 最重要的设计考虑是从输出及内部驱动器获得适当的上升时间和使峰值电流最小化。

强烈建议使用转换速率控制电路以获得合适的$\frac{\mathrm{d}i}{\mathrm{d}t}$转换特性，这可以通过慎重选择驱动器规格来实现。

9.6.3.5 时钟产生及分配

对于时钟馈入全部模块或一组模块的情况，当不需要时应关闭时钟及振荡器。一个好的办法是宁可支持各种低功耗模式，而不是将时钟限制在较低的频率或完全关闭它。

前面所述的扩频时钟（SSC），即“时钟抖动”是减少 EMI 非常有效的办法。抖动本质上对高次谐波更有效，而对低次谐波效果较差。由于频率偏移的绝对值随着谐波数线性增加，因此频谱能量在高次谐波上传播的范围更大，而测量此频谱能量的滤波器宽度是固定的。幸运的是，高频正是某些应用出现最严重错误问题的地方。由于某些元件（如某些打印机的打印电缆）独特的共振条件造成一些应用中低频辐射正好是主要的噪声源，这时抖动的效果可能就不大了。

另一种有效的技术是使用非重叠时钟来处理 EMI。

图 9.17 所示为非重叠时钟，即，具有非一致边沿转换的时钟。从系统观点看，非重叠时钟脉冲边沿通过多时钟系统的连续脉冲边沿之间的过渡时间，有助于消除竞态条件和亚稳态问题。

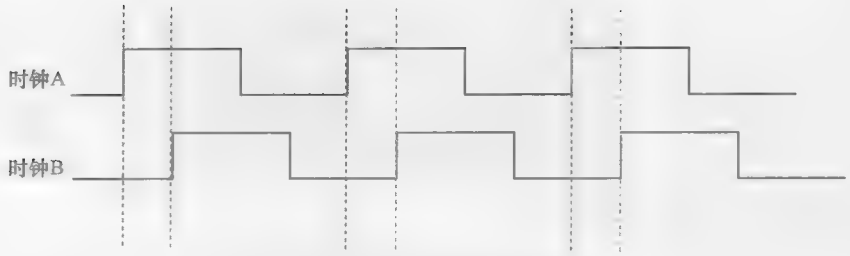


图 9.17 非重叠时钟脉冲边沿

从 EMC 的角度看，加入时钟边沿之间的过渡时间会降低可观察到的峰值电流和电流谐波的峰值幅度。平均电流从时间跨度上来看将保持大致相同，但幅度和频谱形状会发生变化。

另一个与时钟相关的技术是调节时钟的上升/下降时间至绝对最小值。有时需要的变化仅仅是在时钟线上增加一个串联电阻。这个电阻与 PCB 上的时钟线与接地层之间的固有电容形成一个简单的电阻 - 电容低通滤波器。

同样重要的是，避免在常用频率下运行时钟，这样会在时钟谐波附近产生共振结构从而增加噪声。

经常可以看到在电路板上多个高速区域中使用频率相同的多个基准时钟的情况。这将严重影响 EMI 性能，因为每个时钟依次成为加性噪声源。一个更好的办法是使用单个基准输入来改进噪声性能。如果单个基准时钟不可行，则强烈建议使用不同频率的参考时钟，以便将 EMI 扩展到多倍频率上。

我们通过一个例子来进一步理解。图 9.18 所示是具有多个基准时钟输入的 SoC，每个时钟输入馈入 SoC 中的一个高速区域。

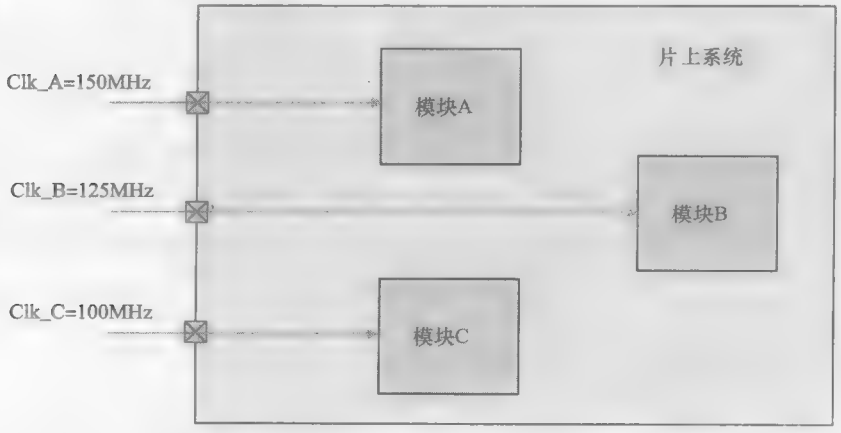


图 9.18 具有不同基准时钟频率的谐波

基准时钟（Clk_A、Clk_B、Clk_C）特意选择不同频率，以便在1.0GHz以下不会出现同频的谐波，Clk_B是8次谐波，而Clk_C是10次谐波。Clk_A与Clk_C在1.2GHz有同频的谐波，即Clk_A的8次谐波和Clk_C的12次谐波。这并不意味着EMI会在几个基频上扩散EMI，以便直到1.2GHz左右谐波噪声才会发生叠加。

9.6.3.6 占空比考虑

一个要考虑的重要问题是，如果占空比正好是50%，复杂梯形开关波形的所有能量都在奇数谐波（1、3、5、7等）。这样，工作在50%的占空比通常是最坏情况。当占空比高于或低于50%时，由于引入了偶次谐波，EMI将按正常情况分布。

9.6.3.7 降低数据总线上的噪声

带宽为8、16、32位甚至更高倍数的数据总线在电路板上跨长距离的情况并不少见，这将导致串扰。

图9.19给出了有助于提高EMC能的总线布局选择。

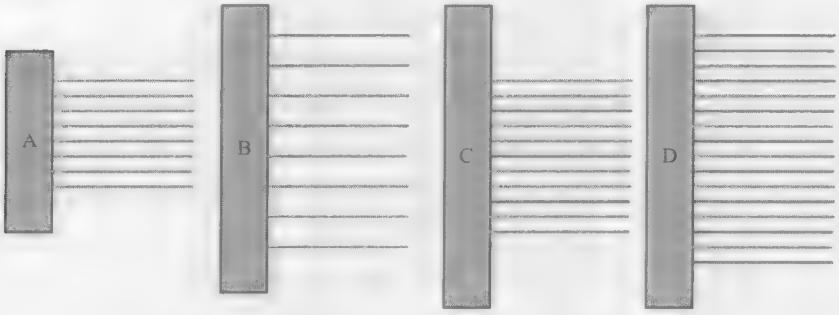


图9.19 有助于提高EMC性能的总线布局选择

“A”是所有8条线紧密排列的典型总线布局，这样就会引入串扰。“B”则通过增加数据线之间的距离来降低噪声。选择“C”是每隔两条数据线插入一条地线的方法，这种方法有可能不可行（如对高速数据线）或如果电路板上空间有限。“D”（接地线与每条数据线交错放置）则对全面降低开关噪声非常有效。

9.6.4 软件层级技术

在硬件层级完全消除瞬态效应尽管令人期待，但这不切实际并且费用

昂贵。本节重点介绍可在微控制器上配置的软件技术，以预防或抑制噪声（如果不能完全消除）。

9.6.4.1 通用 I/O 引脚保护

如图 9.20 所示，所有常规的 I/O 引脚必须在内部有到地和电源的 ESD 保护二极管。流过器件的最大电流必须限定在数据手册中的“绝对最大额定参数”范围内，否则它将会危害和损坏器件。

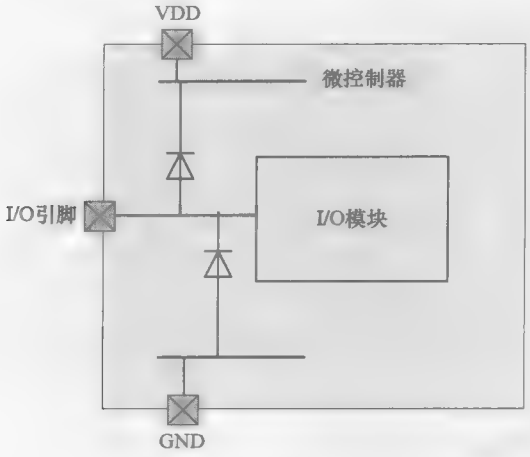


图 9.20 I/O 引脚保护

EMI 和 ESD 控制器件必须提供要求的保护等级而不会减弱输入信号或降低接收电路的特性以至于超过产品规范的要求。对于具有在瞬态波形的噪声带宽之外工作带宽的电路，可以使用低通滤波器、高通滤波器或带通滤波器来实现保护。对于输入的标准保护是低通滤波器，如图 9.21 所示。

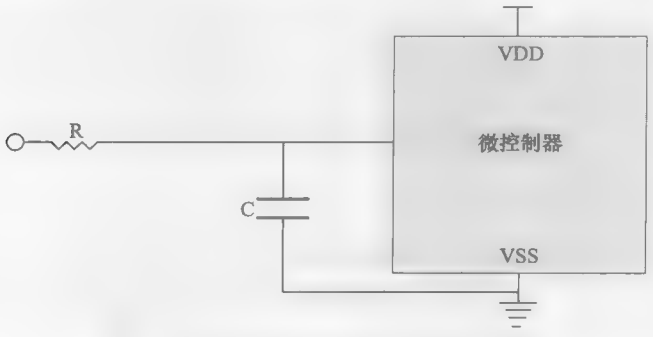


图 9.21 带有低通滤波器的输入引脚保护

串联电阻限制了注入电流。并联电容器把瞬态电流分流至接地系统，因为它要保持电压在稳态值。电阻和电容的数值可根据对输入信号的最大保护或最小影响而变化。

总之，当把系统中几个引脚连接至同一点时，将 I/O 引脚设计为漏极开路有利的。当然，在设计软件时必须注意任何时间只有一个引脚可作为输出，以避免输出驱动器竞争。可行的办法是将这些引脚配置为输出漏极开路，以降低电流竞争的风险。

9.6.4.2 数字输入引脚

针对系统中数字输入比较脆弱的情况，使用软件滤波技术是很重要的，它可以消除由于外部噪声导致的输入引脚短时脉冲波干扰。

可以用简单的技术在输入端读取预定的次数，在大部分时间读到的逻辑状态可认为是正确的状态。

涉及滤波的其他技术是，如果输入的变化持续时间小于阈值则过滤掉该输入信号。这项技术在 IRQ 引脚或键盘中断（KBI）引脚等中断输入中特别有用。它还常用在消抖机械开关输入中（详情参见 9.6.2.7 节）。

9.6.4.3 数字输出和关键寄存器

用户软件应当经常更新输出和关键寄存器，它们控制输出引脚以确保任何轻微故障都会纠正而不发生大问题。这些更新包括：

- 数据定向寄存器
- 可通过软件修改的 I/O 模块
- 用于关键应用的 RAM 寄存器

这些寄存器的更新应当尽可能定期进行。输出及 RAM 寄存器的可靠性不应当受经常写入/更新的影响。应当注意，诸如串行通信、定时器等功能在重新初始化时必须确保在非活动状态，因为一些状态位容易受到相应控制寄存器写入的影响。

9.6.4.4 复位引脚保护

在大多数微控制器中，在调试或编程时复位引脚是置高的（对低电平有效的复位来说），因此从地到复位引脚只需要保护二极管（见图 9.22）。

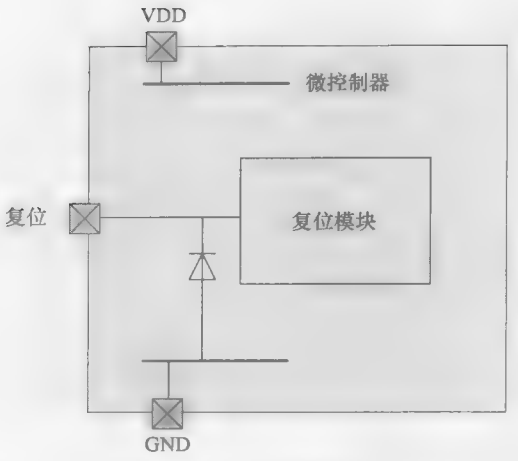


图 9.22 复位引脚保护

在正常工作模式中，复位是从外部驱动的，因此需要另一级别的保护，如图 9.23 所示。

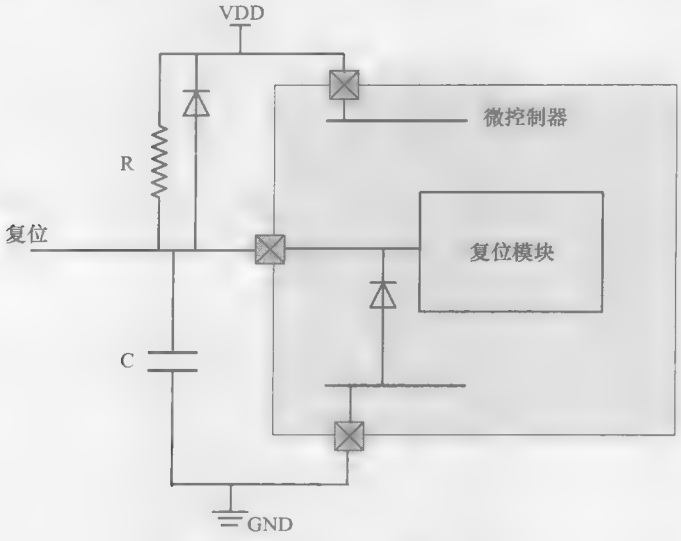


图 9.23 复位引脚保护

电容器帮助吸收瞬变电压，尽管 RC 网路的实际作用是复位开关消抖和设置线路保持为逻辑零的最短时间。

内部和外部二极管将引脚电压箝位在 GND - 0.7V 至 VDD + 0.7V 之间。

9.6.4.5 振荡器和其他敏感引脚

微控制器上最脆弱的引脚通常是高阻抗模拟引脚，如用在振荡器电路、锁相环和模拟信号输入中的引脚。特别要注意，在设计电路板布局时，让这些引脚远离噪声。然而，类似于先前讨论的用于数字引脚的滤波技术，也可应用于某些模拟信号输入引脚，如馈入模－数转换器（ADC）的引脚。在这种情况下，分析转换值可确定这些值是否在预期范围内；通过对所有有效转换简单求平均数，可以降低大多数噪声的影响。

高频振荡器是相当精密的器件，因此对外部噪声很敏感。另外，与其他 I/O 引脚相比，振荡器引脚通常对 ESD 更敏感。

9.6.4.6 看门狗定时器

对任何受到噪声影响的系统，可能会出现代码跑飞而使系统处于未知状态的情况。一个设计良好的看门狗定时器应当能够将系统带回安全状态。

有关任务和严格安全性应用的代表性例子是航天器的推力控制。在太空执行的其中一个最细致的操作是两个航天器的对接。精确的方向控制和动作保证两个舱体准确地排列，以实现对接。航天器推进器的控制系统必须完美无瑕地工作。太空中的物体也会遭受严重的未知噪声干扰。推进器电子控制单元的软件崩溃会导致推进器喷火时间过长或喷火角度错误或两者兼有，从而导致航天器相撞而不是实现对接。因此，在适当的位置上应当有安全装置可探测故障，并在推进器发生不可预见的连续喷火前将 ECU 置于安全状态。

另一个关键应用是外科手术中的机械手，这在先进医疗设备中越来越常见。这些系统能增强医生用最小干预就可以执行复杂手术的能力。在一个手术中，医生开始一个特定的程序，如在一个重要器官中的细小切口，然后就完全控制机械手来运用手术刀。系统中的噪声可能会导致代码跑飞和软件故障，导致机械手不可预见地工作，对病人构成危险。系统必须从这类不需要的噪声引起的故障中恢复正常，从而控制机械手正确操作。

以下是应当遵守的一系列好的做法，以确保硬件能从跑飞的代码状态中迅速而可靠地恢复正常[6]。

- 看门狗定时器的宽度应当能够覆盖系统中所有可用时钟源的超时时间范围。建议尽可能使用最短的看门狗超时周期以确保代码跑飞状态不会持续太久。应用程序本身将规定实际的 COP 超时周期

选择。

- 看门狗定时器的时钟源应独立于其监控的系统的时钟源。更好的应当是给看门狗提供专用的时钟源，如 RC 振荡器。这意味着即使系统时钟由于某种原因停止，导致系统挂起，看门狗定时器仍能超时并复位系统。
- 看门狗向系统发出故障信号的方法必须是能自身容错的。
- 看门狗的关键控制和配置寄存器位应当有写保护，这样一旦设置好就不会被意外地修改。
- 刷新看门狗的方法应当是使跑飞的代码意外刷新看门狗的机会最小。如果跑飞的代码通过某些不可思议的机会刷新看门狗，看门狗将要么不知道跑飞的代码的情况要么很长时间以后才知道。建议不要仅凭 RAM 中的一个字节或一位或者状态寄存器中的一位就刷新看门狗。在刷新看门狗前应当对系统状态进行全面检查。
- 看门狗探测失控状态的反应应当是迅速的。如果看门狗要花太多时间复位系统，则处于未知状态的系统将可能对重要的安全性应用造成许多危害。回忆机械手的例子，机械手发生故障到停止工作的时间越长，病人的生命就越危险。
- 看门狗的正常运行应当是可测试的，以便确保启动后它的功能正常。测试不应该花费过多的时间。
- 看门狗应当能帮助诊断导致看门狗超时的故障。
- 对于软件实现（如果使用噪声关键应用程序则不推荐），应避免将看门狗刷新放在中断程序中。即使 CPU 陷入主程序的未知循环中，中断仍能提供服务。
- 任何服务于看门狗的循环应当能在有限时间内给出超时状态。时间取决于系统能承受 CPU 错误执行代码的时间。

强烈建议看门狗符合 IEC 60730 标准，这是家用电器的安全性标准，可确保操作安全可靠。

IEC 60730 论述了有关交流电应用的机械性能、电气性能、电子性能、使用环境、耐用性、EMC、异常运行等情况。

IEC 60370 将自动控制产品分为三级。

级别 A：不要试图依赖设备的安全性。

级别 B：防止受控设备的不安全操作。

级别 C：防止特殊危害。

遵守 IEC 的安全性标准具有很大的好处，它对噪声环境提供了更好的

抗干扰性，这也是现在测量、工业等其他应用中参考这些标准的原因。

9.6.4.7 非法指令和非法地址复位

迅速从跑飞代码状态中恢复系统的另一个可能途径是在出现非法指令/非法地址时产生复位。非法地址复位是对内存较小的微控制器是最有效的，因为这些微控制器更可能遇到跑飞的代码落到其内存映射中没有实现的部分。伴随着这些中断或复位情况，许多微控制器装有复位状态寄存器和中断状态寄存器，它们有助于确定复位来源或中断来源，使得软件能采取适当的行动。

9.6.4.8 低电压检测/低电压警告

低电压检测（Low Voltage Detect，LVD）或低电压警告（Low Voltage Warning，LVW）增加了器件的敏感性，对电源线（VDD）的电气干扰和传导噪声提供了更好的抗干扰性。

当芯片电源（VDD）低于最小工作电压时，微控制器的性能就无法保证，就没有足够的电力来解码/执行指令和（或）读取内存。最坏的情况是，如果让芯片在低于最小保证电压下的情况工作，则写入内存或寄存器位会导致数据损坏。强烈建议微控制器在这种状态下应自动重置以防止不可预测行为。

如图9.24所示，低电压检测（LVD）功能应在VDD电源低于V_{fall}（LVD）时产生静态复位。在上电期间，微控制器应保持在复位状态直到电压上升到V_{rise}（LVD）之上，从而保证上电和断电时的安全。

值得注意的是，电压下降的V_{fall}（LVD）参考值要一直低于加电电压V_{rise}（LVD）参考值，以避免MCU开始运行并吸收电流时产生的寄生复位。

LVD阈值（升或降）应当是可编程的，以为应用提供充分的灵活性。LVD还允许器件在没有任何外接复位电路时使用，以节省电路板成本。

低电压警告（Low Voltage Warning，LVW）通过确保微控制器在电源受外部噪声干扰时安全运行来进一步改进抗干扰性。在LVD产生系统复位前，LVD能让系统产生预先警告（通过产生一个中断）。

LVW的工作情况如图9.24所示。在电压下降期间，当电压低于V_{fall}（LVW）时，则产生中断，用户能在中断程序中关闭应用直到电源达到器件的正常电压。

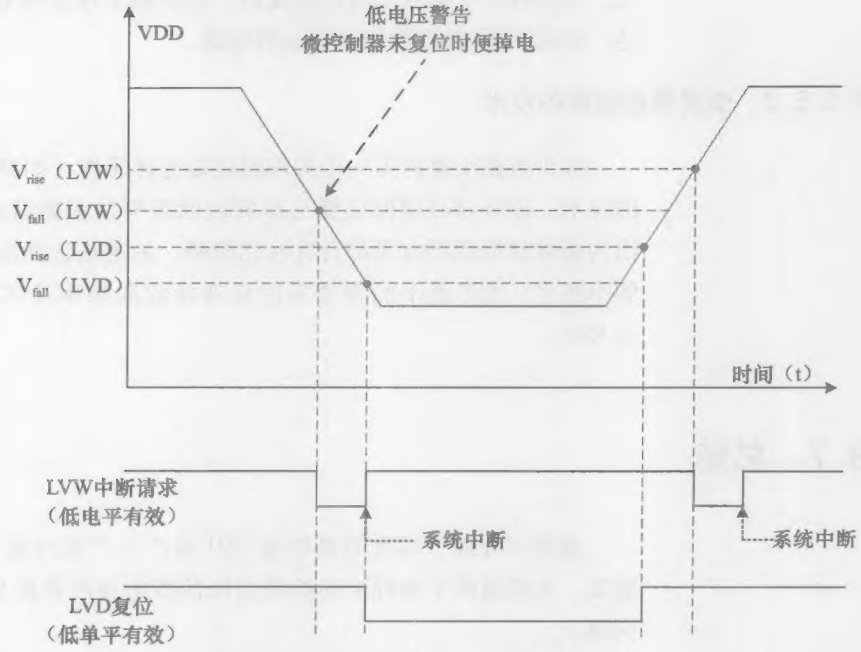


图 9.24 芯片电源的 LVD/LVW 监控

由于 LVD 中所述的类似原因，在电压下降期间 LVW 触发电压“V_{fall}（LVW）”一直低于上电期间的 LVW 跳变电压“V_{rise}（LVW）”。

系统应当允许用户对 LVD 和 LVW 的触发范围进行编程以取得最大灵活性，而不是使用 LVD 和 LVW 固定跳变点。例如，在电压降到 LVD 跳变点和系统复位前，某些应用可能想在 LVW 中断程序中将重要数据由内部存储器保存至外部 EEPROM（电可擦只读存储器）中。编程时把 LVW 和 LVD 跳变点之间的时间多留一点可使得系统有足够的时间来将所有数据存储到 EEPROM 并采取必需的行动用于安全恢复。有很多其他的例子可说明这点是非常有用的。

9.6.5 其他技术

9.6.5.1 多电源和接地引脚

相邻的接地和电源引脚、多个接地和电源引脚、将电源和接地引脚放在中心都有助于将电源与接地电流路径间的互感最大化，并将自感最小

化，可以减少电源电流的回路面积，使去耦工作更有效，如9.6.2.7节所述。这减少了EMC和接地抖动的问题。

9.6.5.2 使用最低频率的技术

减少电磁干扰的另一个关键因数是选择元件（如存储器）的最低频率的技术。设计者必须保证要选择的较低频率技术能满足时序和性能目标，因为通常较低频率技术都会有性能限制。关键是选择满足性能指标的较低频率技术，而不是不加细致考虑就选择较高频率技术。频率越低越能减少EMI。

9.7 总结

如果不处理，多余的辐射或EMI将产生严重问题。设计好的EMC很重要。本章提供了多种不同抽象层次的指南来改善图9.25中总结的EMC问题。

然而，必须注意，通常物美价廉的选择是在源头处理问题。

图9.25 影响瞬态干扰的因素

迈向数字电路设计系统工程师必读的一本书

本书的主要内容涉及时钟和复位、多时钟域设计、时钟分频器、低功耗设计技术、流水线技术、字节顺序、消抖技术和电磁兼容性等方面。第1章介绍亚稳态的相关内容。第2章介绍同步设计的时钟技术，并提出可行的时钟方案，此外也介绍了系统复位策略。第3章介绍多时钟设计的问题和处理方法，几种可能的跨时钟域情况和跨时钟域数据传输方法等。第4章介绍奇数、偶数、小数分频电路的实现和对优缺点的权衡。第5章介绍数字电路功耗来源，并分别从系统级、体系结构级、寄存器传输级和晶体管级提出一系列降低功耗的方法。第6章介绍流水线的基本原理、性能的增加和导致的冒险。第7章讨论使用最佳字节顺序的方法。第8章介绍典型的开关行为和软硬件消抖技术。第9章介绍电磁干扰的原理、规程、标准和认证，电磁干扰的影响因素和减少电磁干扰的方法。

本书特点：

- 本书实践性强，摆脱了繁杂的公式，从工程角度对技术原理进行解释，便于读者掌握技术，同时又能使读者从一定深度理解和应用技术。
- 每章一专题，各成体系，同时全书系统性很强，能使读者从整体上建立数字电路设计的知识架构，在设计电路时从各个角度实现功耗、面积和性能的权衡优化，满足设计目标。
- 涵盖EMC、低功耗技术（如DVFS技术）、消抖等专题性内容，这些内容在其他书籍中鲜有提到。
- 配图丰富，不仅有利于初学者学习，而且适合中高级读者查阅和深入理解，快速应用。

作者简介：

Mohit Arora 现供职于Freescale半导体公司，任高级系统工程师。从2005年以来，他一直从事IP/SoC架构设计，负责设计和开发了面向众多市场的SoC产品。现在作为一名系统工程师，他的职责主要是参与产品的定义，规格书的撰写。他研发的产品既有面向中高端工业应用的MCU，也有面向消费电子市场的MPU。在加入Freescale公司之前，他曾经供职于安捷伦、意法半导体以及DCM等公司，专注于USB 2.0 PHY、PCI-Express、Infiniband和串行ATA协议等技术领域。

他于2000年在印度NSIT（Netaji Subhas Institute of Technology）获得电子与通信学士学位，他在国际学术刊物上发表了30多篇论文，并拥有串行链路方面的一项专利。

Springer

投稿热线：(010) 88379604
客服热线：(010) 88378991 88361066
购书热线：(010) 68326294 88379649 68995259

华章网站：www.hzbook.com
网上购书：www.china-pub.com
数字阅读：www.hzmedia.com.cn

上架指导：集成电路设计

ISBN 978-7-111-44939-3

9 787111 449393 >

定价：59.00元

[General Information]
=
=204
SS =13498412